剖析各类驱动的实现过程，分享移动开发的移植技巧



高金昌　张明星◎编著

Android
底层驱动分析和移植

- 贯通底层驱动、中间层JNI制作、上层UI 接口设计。
- 面向实战，深入剖析各驱动系统的完整实现流程。
- 源码分析+全真示例+图片解析=更易于理解的思维路径。
- 教授精髓，精讲精炼。赠送源码，拿来就用。

清华大学出版社

Android底层驱动分析和移植

高金昌　张明星　编著

清华大学出版社
北　京

内 容 简 介

Android 系统从诞生到现在，在短短的几年时间里，便凭借其操作易用性和开发的简洁性，赢得了广大用户和开发者的支持。本书内容分为 3 篇，共 22 章，循序渐进地讲解了 Android 底层系统中的典型驱动方面的知识。本书从获取源码和源码结构分析讲起，依次讲解了基础知识篇、Android 专有驱动篇和典型驱动移植篇 3 部分的基本知识。在讲解每一个驱动时，从 Android 系统的架构开始讲起，从内核分析到具体的驱动实现，再从 JNI 层架构分析到 Java 应用层的接口运用，最后到典型驱动系统移植和开发，彻底剖析了每一个典型驱动系统的完整实现流程。本书几乎涵盖了所有 Android 底层驱动的内容，讲解方法通俗易懂，内容翔实，不但适合应用高手的学习，也特别有利于初学者学习和消化。

本书适合作为 Android 驱动开发者、Linux 开发人员、Android 底层学习人员、Android 爱好者、Android 源码分析人员、Android 应用开发人员的学习用书，也可以作为相关培训学校和大专院校相关专业的教学用书。

图书在版编目（CIP）数据

Android 底层驱动分析和移植/高金昌，张明星编著．—北京：清华大学出版社，2015
ISBN 978-7-302-39745-8

I．①A…　II．①高…　②张…　III．①移动终端-应用程序-程序设计　IV．①TN929.53

中国版本图书馆 CIP 数据核字（2015）第 071318 号

责任编辑：朱英彪
封面设计：刘　超
版式设计：魏　远
责任校对：王　云
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：203mm×260mm　**印　　张：**50.75　**字　　数：**1530 千字
版　　次：2015 年 7 月第 1 版　**印　　次：**2015 年 7 月第 1 次印刷
印　　数：1～3000
定　　价：96.00 元

产品编号：061538-01

前　　言

2007 年 11 月 5 日，Google 公司宣布的基于 Linux 平台的开源手机操作系统 Android 诞生，该平台号称是首个为移动终端打造的真正开放和完整的移动软件。本书将引领广大读者一起探讨这款系统的神奇之处。

市场占有率高居第一

截至 2014 年 1 月，Android 在手机市场上的占有率从 2013 年的 68.8%上升到 78.9%。而 iOS 则从 2013 年的 19.4%下降到 15.5%，WP 系统从原来的 2.7%小幅上升到 3.6%。

从数据上来看，Android 平台占据了市场的主导地位，继续称当老大的角色。Android 市场的占有率增加幅度较大，WP 市场小幅增长，但 iOS 却有所下降。就目前来看，智能手机的市场已经饱和，大多数人都在各个平台中转换。而就在这样的市场背景下，Android 还增长了 10%左右的占有率，确实不易。

为开发人员提供了滋长的“沃土”

（1）保证开发人员可以迅速转型为 Android 应用开发

Android 应用程序是通过 Java 语言开发的，只要具备 Java 开发基础，就能很快地上手并掌握。作为单独的 Android 应用开发，对 Java 编程门槛的要求并不高，即使是没有编程经验的门外汉，也可以在突击学习 Java 之后学习 Android。另外，Android 完全支持 2D、3D 和数据库，并且和浏览器实现了集成。所以通过 Android 平台，程序员可以迅速、高效地开发出绚丽多彩的应用，例如常见的工具、管理、互联网和游戏等。

（2）定期召开奖金丰厚的 Android 大赛

为了吸引更多的用户使用 Android 开发，Google 公司已经成功举办了奖金为数千万美元的开发者竞赛，鼓励开发人员创建出创意十足、十分有用的软件。这种大赛对于开发人员来说，不但能练习自己的开发水平，并且高额的奖金也是学员们学习的动力。

（3）开发人员可以利用自己的作品赚钱

为了能让 Android 平台吸引更多的关注，Google 公司提供了一个专门下载 Android 应用的门店：Android Market，地址是 https://play.google.com/store。这个门店允许开发人员发布应用程序，也允许 Android 用户下载获取自己喜欢的程序。作为开发者，需要申请开发者账号，申请后才能将自己的程序上传到 Android Market，并且可以对自己的软件进行定价。只要你的软件程序足够吸引人，你就可以获得很好的金钱回报，这样实现了程序员学习和赚钱两不误，因此吸引了更多的开发人员加入 Android 大军中来。

本书的内容

本书内容分为 3 篇，共 22 章，循序渐进地讲解了 Android 底层系统中的典型驱动方面的知识。本书从

获取源码和源码结构分析讲起，依次讲解了基础知识篇、Android 专有驱动篇和典型驱动移植篇 3 部分的基本知识。在讲解每一个驱动时，从 Android 系统的架构开始讲起，从内核分析到具体的驱动实现，再从 JNI 层架构分析到 Java 应用层的接口运用，最后到典型驱动系统移植和开发，彻底剖析了每一个典型驱动系统的完整实现流程。本书几乎涵盖了所有 Android 底层驱动的内容，讲解方法通俗易懂，内容翔实，不但适合应用高手们的学习，也特别有利于初学者学习和消化。

本书的版本

Android 系统自 2008 年 9 月发布第一个版本 1.1 以来，截至 2014 年 10 月发布最新版本 5.0，一共存在十多个版本。由此可见，Android 系统升级频率较快，一年之中最少有两个新版本诞生。如果过于追求新版本，会造成力不从心的结果。所以在此建议广大读者不必追求最新的版本，只需关注最流行的版本即可。据官方统计，截至 2014 年 10 月 25 日，占据前 3 位的版本分别是 Android 4.4、Android 4.3 和 Android 4.2。其中从 Android 4.4 开始，Android L 和 Android 5.0 的底层架构知识基本类似。其实这 3 个版本的区别并不是很大，只是在顶层 API 应用层的细节上进行了更新。为了及时学习 Android 系统的最新功能，本书中使用的版本是目前（本书成稿时）最新的 Android 5.0。

本书特色

本书内容十分丰富，分析细致、精准、全面。我们的目标是通过一本图书，提供多本图书的价值，读者可以根据自己的需要有选择地阅读。在内容的编写上，本书具有以下特色。

1．内容全面，讲解细致

本书几乎涵盖了 Android 系统中所有的独有驱动和设备驱动，详细讲解了每一个典型驱动的实现过程和具体移植方法。每一个知识点都力求用通俗易懂的语言详尽展现在读者面前。

2．遵循合理的主线进行讲解

为了使广大读者彻底了解 Android 平台中的各个驱动系统，在讲解每一个驱动系统时，从 Linux 内核开始讲起，依次剖析了驱动层实现、JNI 层分析、Java 应用和系统移植改造等内容，遵循了从底层到顶层，实现了驱动系统大揭秘的目标。

3．章节独立，自由阅读

本书中的每一章内容都可以独自成书，读者既可以按照本书编排的章节顺序进行学习，也可以根据自己的需求对某一章节进行针对性的学习。与传统的计算机书籍相比，阅读本书会更轻松。

4．驱动典型，实用性强

本书讲解了现实中最典型驱动系统的实现和移植知识，这些驱动都是在商业项目中最需要的部分，读者可以直接将本书中的知识应用到自己的项目中，实现无缝对接。

读者对象

本书适合作为 Android 驱动开发者、Linux 开发人员、Android 底层学习人员、Android 爱好者、Android 源码分析人员、Android 应用开发人员的学习用书，也可以作为相关培训学校和大专院校相关专业的教学用书。



参与本书编写的人员还有周秀、付松柏、邓才兵、钟世礼、谭贞军、张加春、王教明、万春潮、郭慧玲、侯恩静、程娟、王文忠、陈强、何子夜、李天祥、周锐、朱桂英、张元亮、张韶青、秦丹枫。本团队在编写过程中，得到了清华大学出版社工作人员的大力支持，正是各位编辑的求实、耐心和效率，才使得本书在这么短的时间内出版。此外也十分感谢我们的家人，在写作时给予了巨大的支持。另外告知各位读者，因编者水平有限，如有纰漏和不尽如人意之处，恳请读者提出意见或建议，以便修订并使之更臻完善。另外我们提供了售后支持网站：http://www.chubanbook.com/和 QQ 群 192153124，读者朋友如有疑问可以在此提出，一定会得到满意的答复。

编 者

目　录

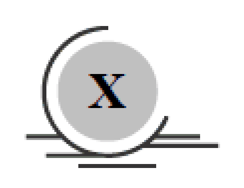
第 1 篇　基础知识篇

第 2 篇 Android 专有驱动篇



第 3 篇　典型驱动移植篇





第 1 篇



基础知识篇

- 第 1 章　Android 底层开发基础
- 第 2 章　Android 驱动开发基础
- 第 3 章　主流内核系统解析

第 1 章　Android 底层开发基础

2007 年，Google 公司推出了一款无与伦比的移动智能设备系统——Android，这是一种建立在 Linux 基础之上为手机、平板等移动设备提供的软件解决方案。截至 2014 年 9 月，根据知名 IDC 公司的统计，Android 系统在世界智能手机发货量中占据 82%的份额，已经成为了当今最受欢迎的智能设备系统之一。本章将引领读者一起来了解 Android 系统基本知识，并详细讲解 Android 底层开发所必须具备的基础知识。

1.1　Android 系统介绍

Android 一词最早出现于法国作家利尔亚当在 1886 年发表的科幻小说《未来夏娃》中，他将外表像人的机器起名为 Android，如图 1-1 所示。



图 1-1　Android 机器人

从 2008 年 HTC 和 Google 联手推出第一台 Android 手机 G1 开始，在 2011 年第一季度，Android 在全球的市场份额首次超过塞班系统，跃居全球第一。2011 年 11 月数据显示，Android 占据全球智能手机操作系统市场 52.5%的份额，中国市场占有率为 58%。如今 Android 已经成为了现在市面上主流的智能手机操作系统，随处都可以见到这个绿色机器人的身影。

Android 机型数量庞大，简单易用，相当自由的系统能让厂商和客户轻松地定制各样的 ROM、各种桌面部件和主题风格。其简单而华丽的界面得到了广大客户的认可，对手机进行刷机也是不少 Android 用户所津津乐道的事情。

Android 版本数量较多，市面上同时存在着 1.6、2.0、2.1、2.2、2.3、4.4.2、5.0 等各种版本的 Android 系统手机，应用软件对各版本系统的兼容性对程序开发人员是一个不小的挑战。同时由于开发门槛低，导致应用数量虽然很多但是应用质量参差不齐，甚至出现不少恶意软件，使一些用户受到损失。同时 Android 没有对各厂商在硬件上进行限制，导致一些用户在低端机型上体验不佳。另一方面，因为 Android 的应用主要使用 Java 语言开发，其运行效率和硬件消耗一直是其他手机用户所非议的地方。

1.2　Android 系统架构介绍

Android 系统是一个移动设备的开发平台，其软件层次结构包括操作系统（OS）、中间件（MiddleWare）和应用程序（Application）。根据 Android 的软件框图，其软件层次结构自下而上分为以下 4 层。

（1）操作系统层（OS）。

（2）各种库（Libraries）和 Android 运行环境（RunTime）。

（3）应用程序框架（Application Framework）。

（4）应用程序（Application）。

上述各个层的具体结构如图 1-2 所示。

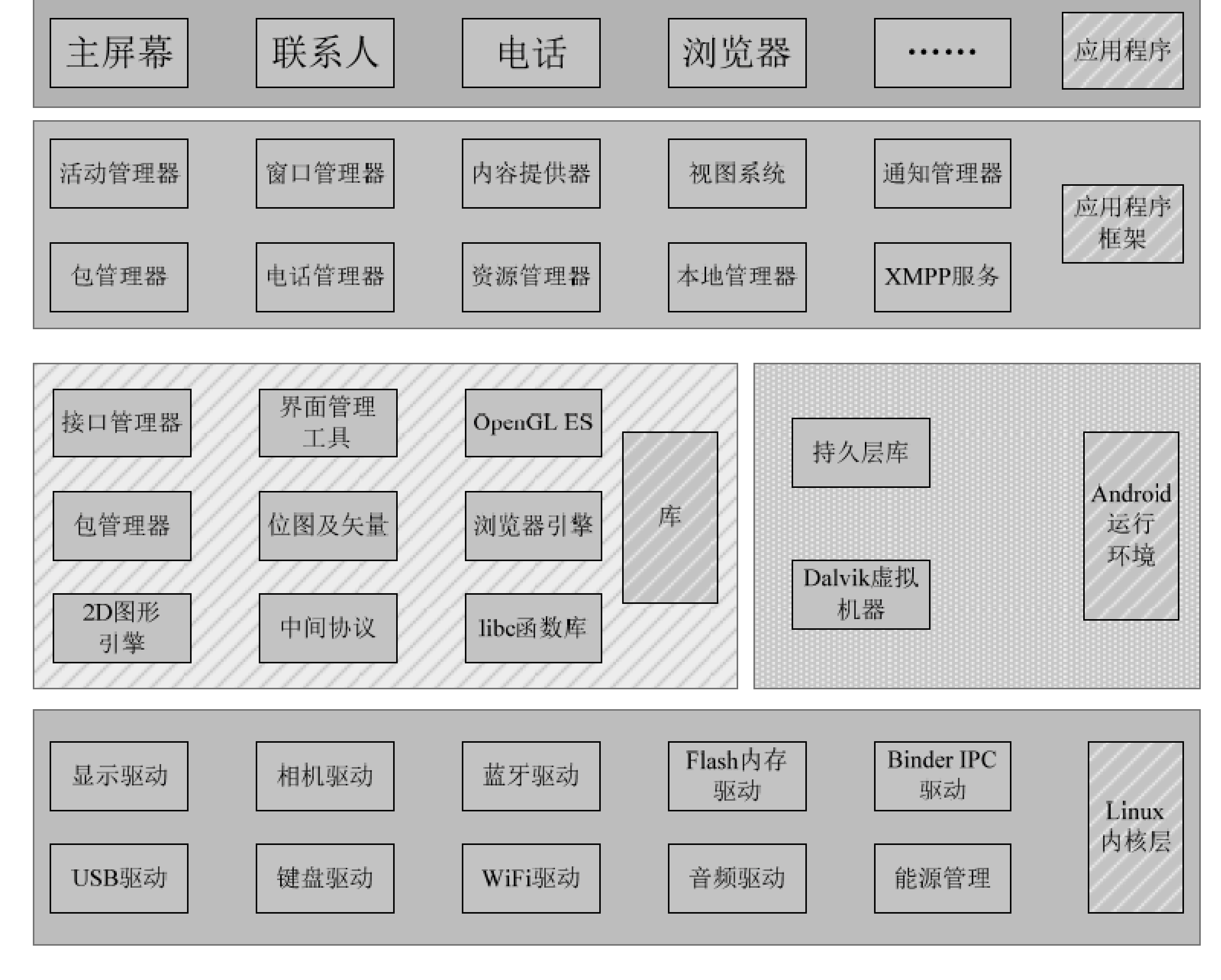


图 1-2　Android 操作系统的组件结构图

本节将详细介绍 Android 操作系统的基本组件结构方面的知识。

1.2.1　底层操作系统层（OS）

因为 Android 源于 Linux，使用了 Linux 内核，所以 Android 使用 Linux 2.6 作为操作系统。Linux 2.6 是一种标准的技术，Linux 也是一个开放的操作系统。Android 对操作系统的使用包括核心和驱动程序两部分，Android 的 Linux 核心为标准的 Linux 2.6 内核，其更需要一些与移动设备相关的驱动程序，主要的驱动如下。

☑　显示驱动（Display Driver）：是常用的基于 Linux 的帧缓冲（Frame Buffer）驱动。
☑　Flash 内存驱动（Flash Memory Driver）：是基于 MTD 的 Flash 驱动程序。
☑　照相机驱动（Camera Driver）：常用基于 Linux 的 V4L（Video for Linux）驱动。
☑　音频驱动（Audio Driver）：常用基于 ALSA（Advanced Linux Sound Architecture、高级 Linux 声音体系）驱动。
☑　WiFi 驱动（Camera Driver）：基于 IEEE 802.11 标准的驱动程序。
☑　键盘驱动（KeyBoard Driver）：作为输入设备的键盘驱动。
☑　蓝牙驱动（Bluetooth Driver）：基于 IEEE 802.15.1 标准的无线传输技术。
☑　Binder IPC 驱动：Android 一个特殊的驱动程序，具有单独的设备节点，提供进程间通信的功能。
☑　Power Management（能源管理）：用于管理电池电量等信息。

1.2.2　各种库（Libraries）和 Android 运行环境（RunTime）

本层次对应一般嵌入式系统，相当于中间件层次。Android 的本层次分成两个部分：一个是各种库，另一个是 Android 运行环境。本层的内容大多是使用 C 和 C++实现的，其中包含了如下各种库。

☑　C 库：C 语言的标准库，也是系统中一个最为底层的库，C 库通过 Linux 的系统调用来实现；

- ☑ 多媒体框架（MediaFrameword）：是 Android 多媒体的核心部分，基于 PacketVideo（即 PV）的 OpenCORE，从功能上本库一共分为两部分，一部分是音频、视频的回放（PlayBack），另一部分则是音视频的记录（Recorder）。
- ☑ SGL：2D 图像引擎。
- ☑ SSL：即 Secure Socket Layer，位于 TCP/IP 协议与各种应用层协议之间，为数据通信提供安全支持。
- ☑ OpenGL ES：提供了对 3D 的支持。
- ☑ 界面管理工具（Surface Management）：提供了对管理显示子系统等功能。
- ☑ SQLite：一个通用的嵌入式数据库。
- ☑ WebKit：网络浏览器的核心。
- ☑ FreeType：位图和矢量字体的功能。

一般情况下，Android 的各种库是以系统中间件的形式提供的，它们的显著特点是与移动设备平台的应用密切相关。另外，Android 的运行环境主要是指 Dalvik（虚拟机）技术。Dalvik 和一般的 Java 虚拟机（Java VM）有如下区别。

- ☑ Java 虚拟机：执行的是 Java 标准的字节码（Bytecode）。从 Android 5.0 开始，ART 将作为应用程序的默认运行环境。而 Java 虚拟机只是作为一个备选项，迟早会退出历史舞台。
- ☑ Dalvik：执行的是 Dalvik 可执行格式（.dex）执行文件。在执行的过程中，每一个应用程序即一个进程（Linux 的一个 Process）。

二者最大的区别在于 Java VM 是基于栈的虚拟机（Stack-based），而 Dalvik 是基于寄存器的虚拟机（Register-based）。显然，后者最大的好处在于可以根据硬件实现更大的优化，更适合移动设备的特点。

1.2.3 应用程序框架（Application Framework）

在整个 Android 系统中，和应用开发最相关的是 Application Framework，在这一层，Android 为应用程序层的开发者提供了各种功能强大的 APIs，这实际上是一个应用程序的框架。由于上层的应用程序是以 Java 构建的，在本层提供了程序中所需要的各种控件，例如 Views（视图组件）、List（列表）、Grid（栅格）、Text Box（文本框）、Button（按钮），甚至还有一个嵌入式的 Web 浏览器。

一个基本的 Andoid 应用程序可以利用应用程序框架中的以下 5 个部分。

- ☑ Activity：活动。
- ☑ Broadcast Intent Receiver：广播意图接收者。
- ☑ Service：服务。
- ☑ Content Provider：内容提供者。
- ☑ Intent and Intent Filter：意图和意图过滤器。

1.2.4 顶层应用程序（Application）

Android 的应用程序主要是用户界面（User Interface）方面的，本层通常使用 Java 语言编写，其中还可以包含各种被放置在 res 目录中的资源文件。Java 程序和相关资源在经过编译后，会生成一个 APK 包。Android 本身提供了主屏幕（Home）、联系人（Contact）、电话（Phone）、浏览器（Browers）等众多的核心应用，同时应用程序的开发者还可以使用应用程序框架层的 API 实现自己的程序，这也是 Android 开源的巨大潜力体现。

1.3　获取 Android 源码

要想进行 Android 底层开发的工作，需要先获取其开源代码。目前市面中的主流操作系统是 Windows、Linux 和 Mac OS。因为 Mac OS 属于类 Linux 系统，所以本节将只介绍在 Windows 系统和 Linux 系统中获取 Android 源码的知识。

1.3.1　在 Linux 系统中获取 Android 源码

在 Linux 系统中，通常使用 Ubuntu 来下载和编译 Android 源码。由于 Android 的源码内容很多，Google 采用了 git 的版本控制工具，并对不同的模块设置不同的 git 服务器，我们可以用 repo 自动化脚本来下载 Android 源码，下面介绍如何一步一步地获取 Android 源码的过程。

（1）下载 repo

在用户目录下，创建 bin 文件夹，用于存放 repo，并把该路径设置到环境变量中，命令如下：

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
```

下载 repo 的脚本，用于执行 repo，命令如下：

```
$ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
```

设置可执行权限，命令如下：

```
$ chmod a+x ~/bin/repo
```

（2）初始化一个 repo 的客户端

在用户目录下，创建一个空目录，用于存放 Android 源码，命令如下：

```
$ mkdir AndroidCode
$ cd AndroidCode
```

进入 AndroidCode 目录，并运行 repo 下载源码，下载主线分支的代码，主线分支包括最新修改的 bug，以及并未正式发出版本的最新源码，命令如下：

```
$ repo init -u https://android.googlesource.com/platform/manifest
```

下载其他分支，正式发布的版本可以通过添加-b 参数来下载，命令如下：

```
$ repo init -u https://android.googlesource.com/platform/manifest -b
                                              android-5.0.0_r1
```

例如可以使用如下命令来初始化最新 Android 源代码。

```
./repo init -u https://android.googlesource.com/platform/manifest -b android-5.0.0_r1
```

输入上面的命令后回车执行，如图 1-3 所示。

```
guan@guan-PC /cygdrive/g/android/bin/aa
$ repo init -u https://android.googlesource.com/platform/manifest -b android-5.0.0_r1
warning: gpg (GnuPG) is not available.
warning: Installing it is strongly encouraged.

Get https://gerrit.googlesource.com/git-repo
remote: Counting objects: 117, done
remote: Finding sources: 100% (117/117)
remote: Total 2955 (delta 1549), reused 2955 (delta 1549)
接收对象中: 100% (2955/2955), 2.51 MiB | 33.00 KiB/s, 完成.
```

图 1-3　选择下载的分支

在下载过程中会需要填写 Name 和 Email，填写完毕之后，选择 Y 进行确认，最后提示 repo 初始化完成，这时可以开始同步 Android 源码了，同步过程很漫长，需要耐心等待，执行下面命令开始同步代码：

```
$ repo sync
```

经过上述步骤后，即开始下载并同步 Android 源码，界面效果如图 1-4 所示。

```
/cygdrive/g/android/bin/aa
 * [新tag]          android-cts-4.2_r1 -> android-cts-4.2_r1
 * [新tag]          android-cts-4.2_r2 -> android-cts-4.2_r2
 * [新tag]          android-cts-4.4_r1 -> android-cts-4.4_r1
 * [新tag]          android-cts-verifier-4.0.3_r1 -> android-cts-verifier-4.0.3_r1
 * [新tag]          android-cts-verifier-4.0_r1 -> android-cts-verifier-4.0_r1
 * [新tag]          android-l-preview_r2 -> android-l-preview_r2
 * [新tag]          android-sdk-4.0.3-tools_r1 -> android-sdk-4.0.3-tools_r1
 * [新tag]          android-sdk-4.0.3_r1 -> android-sdk-4.0.3_r1
 * [新tag]          android-sdk-4.4.2_r1 -> android-sdk-4.4.2_r1
 * [新tag]          android-sdk-4.4.2_r1.0.1 -> android-sdk-4.4.2_r1.0.1
 * [新tag]          android-sdk-adt_r16.0.1 -> android-sdk-adt_r16.0.1
 * [新tag]          android-sdk-adt_r20 -> android-sdk-adt_r20
 * [新tag]          android-sdk-support_r11 -> android-sdk-support_r11
Fetching projects:   4% (20/482)  Fetching project platform/dalvik
 25  328M   25 82.2M    0     0   100k      0  0:55:39  0:13:54  0:41:45 75486curl: (35) Unknow
n SSL protocol error in connection to android.googlesource.com:443
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100     1    0     1    0     0      0      0 --:--:--  0:00:01 --:--:--     0
  0     0    0     0    0     0      0      0 --:--:--  0:00:05 --:--:--     0remote: Sending a
pproximately 114.87 MiB ...
remote: Counting objects: 4376, done
remote: Finding sources: 100% (84/84)
  0     0    0     0    0     0      0      0 --:--:--  0:00:06 --:--:--     0接收对象中:  0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:07 --:--:--     0接收对象中:  0%
 25  328M   25 83.9M    0     0   101k      0  0:55:11  0:14:04  0:41:07  192k接收对象中:  0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:10 --:--:--     0接收对象中:  0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:11 --:--:--     0接收对象中:  0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:12 --:--:--     0接收对象中:  0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:13 --:--:--     0接收对象中:  0%
 25  328M   25 84.8M    0     0   102k      0  0:55:00  0:14:10  0:40:50  157k接收对象中:  0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:15 --:--:--     0接收对象中:  0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:16 --:--:--     0接收对象中:  0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:17 --:--:--     0接收对象中:  0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:18 --:--:--     0接收对象中:  0%
  0     0    0     0    0     0      0      0 --:--:--  0:00:19 --:--:--     0接收对象中:  1%
 25  328M   25 85.3M    0     0   101k      0  0:55:03  0:14:16  0:40:47 76923接收对象中:  1%
  0     0    0     0    0     0      0      0 --:--:--  0:00:21 --:--:--     0接收对象中:  1%
 26  328M   26 85.5M    0     0   102k      0  0:55:01  0:14:18  0:40:43 88897接收对象中:  1%
  0     0    0     0    0     0      0      0 --:--:--  0:00:23 --:--:--     0接收对象中:  1%
  0     0    0     0    0     0      0      0 --:--:--  0:00:24 --:--:--     0接收对象中:  1% (8150/462517)
  0     0    0     0    0     0      0      0 --:--:--  0:00:25 --:--:--     0接收对象中:  1% (8855/462517)
  0     0    0     0    0     0      0      0 --:--:--  0:00:26 --:--:--     0接收对象中:  2% (9251/462517)
 26  328M   26 86.1M    0     0   102k      0  0:54:56  0:14:23  0:40:33  139k
```

图 1-4　正在下载源码

因为网络方面的原因，可能执行“./repo init -u https://android.googlesource.com/platform/manifest-b android-5.0.0_r1”初始化命令会失败，提示一些类似网络连接失败的信息，此时不用理会，只需继续执行这个命令。如果出现多次失败提示，则可以尝试使用以下方法来解决。

（1）使用如下命令删除 Android 5.0 文件中的缓存文件，然后重新执行初始化命令。

```
rm -rf * -R
```

（2）隔一段时间或者晚上、凌晨的时候下载，一般这个时候的网络环境容易下载 Android 源代码。如果看到类似图 1-5 所示的信息，则表示连接成功，正在初始化。

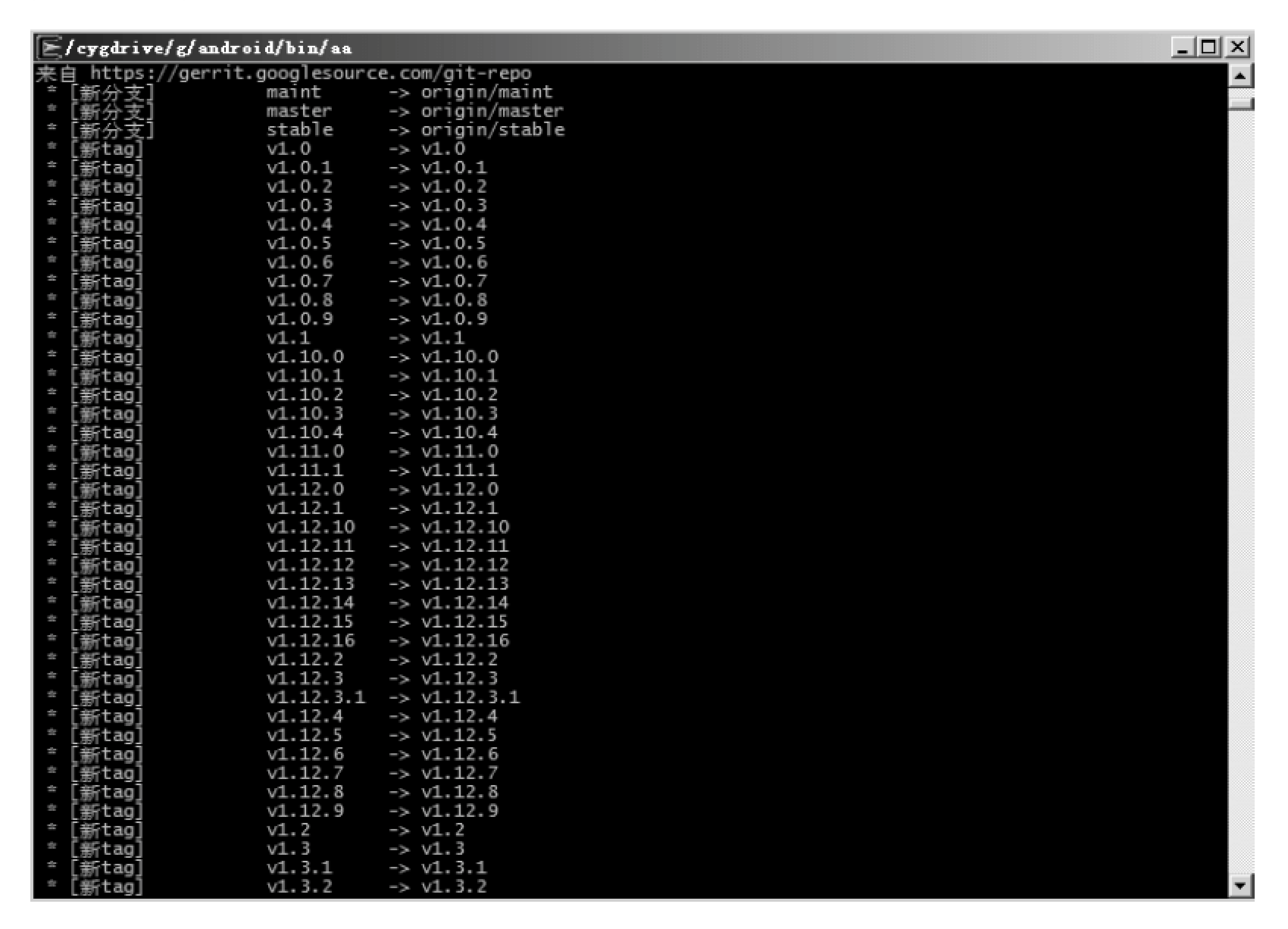


图 1-5　成功初始化



注意：

（1）在源代码下载过程中，在源代码下载目录看不到任何文件，打开“显示/隐藏”，会看到一个名为“.repo”的文件夹，这个文件夹是用来保存 Android 源代码的临时文件。

（2）当文件最后下载接近完成时，会从“.repo”文件夹中导出 Android 源代码。

（3）当 Android 5.0 源代码下载完成后，我们可以看到 Android 源代码下载目录中会有 bionic、bootable、build、cts、dalvik 等文件夹目录，这些就是 Android 的源代码。

（4）如果不得不关闭电脑停止下载，可以在源代码下载的终端中按下 Ctrl+C 或者 Ctrl+Z 快捷键停止源代码的下载，这样不会造成源代码的丢失或损坏。

1.3.2　在 Windows 平台上获取 Android 源码

在 Windows 平台上获取 Android 源码的方式和在 Linux 中的获取原理相同，但是需要预先在 Windows 平台上面搭建一个 Linux 环境，此处需要用到 cygwin 工具。cygwin 的作用是构建一套在 Windows 上的 Linux 模拟环境，下载 cygwin 工具的地址为 http://cygwin.com/install.html。

下载成功后会得到一个名为 setup.exe 的可执行文件，通过此文件可以更新和下载最新的工具版本，具体流程如下。

（1）启动 cygwin，如图 1-6 所示。

（2）单击“下一步”按钮，在弹出的对话框中选中第一个单选按钮，表示从网络下载安装，如图 1-7 所示。



图 1-6　启动 cygwin

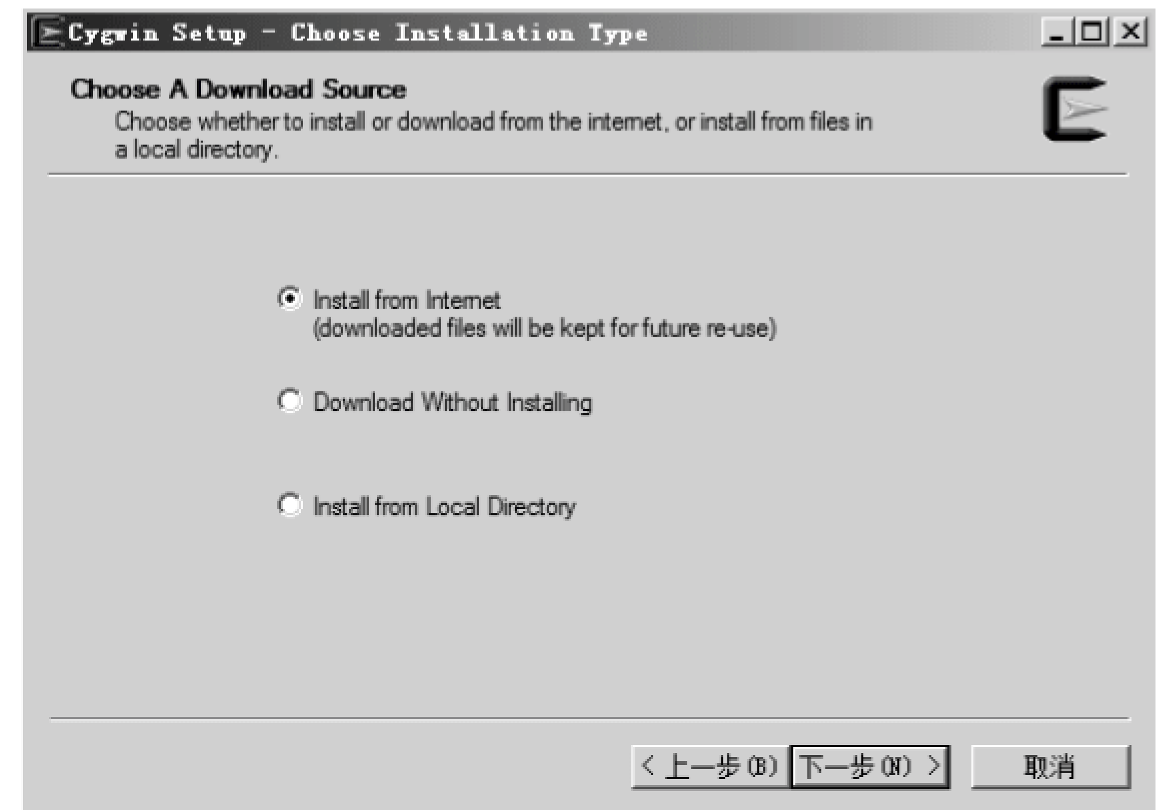


图 1-7　选择从网络下载安装

（3）单击“下一步”按钮，在弹出的对话框中选择安装根目录，如图 1-8 所示。

（4）单击“下一步”按钮，在弹出的对话框中选择临时文件目录，如图 1-9 所示。

（5）单击“下一步”按钮，在弹出的对话框中设置网络代理。如果所在网络需要代理，则在这一步进行设置，如果不用代理，则选择直接下载，如图 1-10 所示。

（6）单击“下一步”按钮，在弹出的对话框中选择下载站点。一般选择离我们比较近的站点速度会比较快，这里选择的是台湾站点，如图 1-11 所示。

（7）单击“下一步”按钮，在弹出的对话框中开始更新工具列表，如图 1-12 所示。

（8）单击“下一步”按钮，在弹出的对话框中选择需要下载的工具包。在此我们需要依次下载 curl、

git、python 这些工具，如图 1-13 所示。

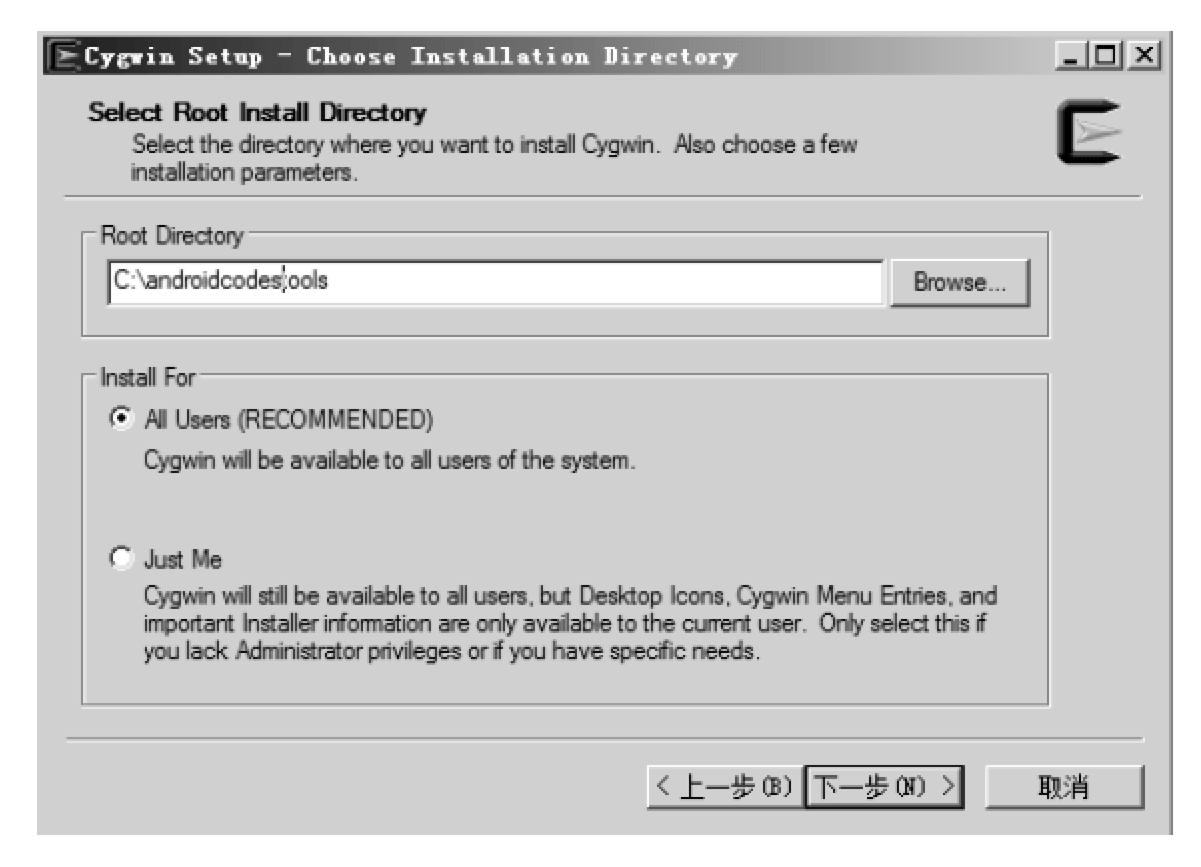


图 1-8　选择安装根目录

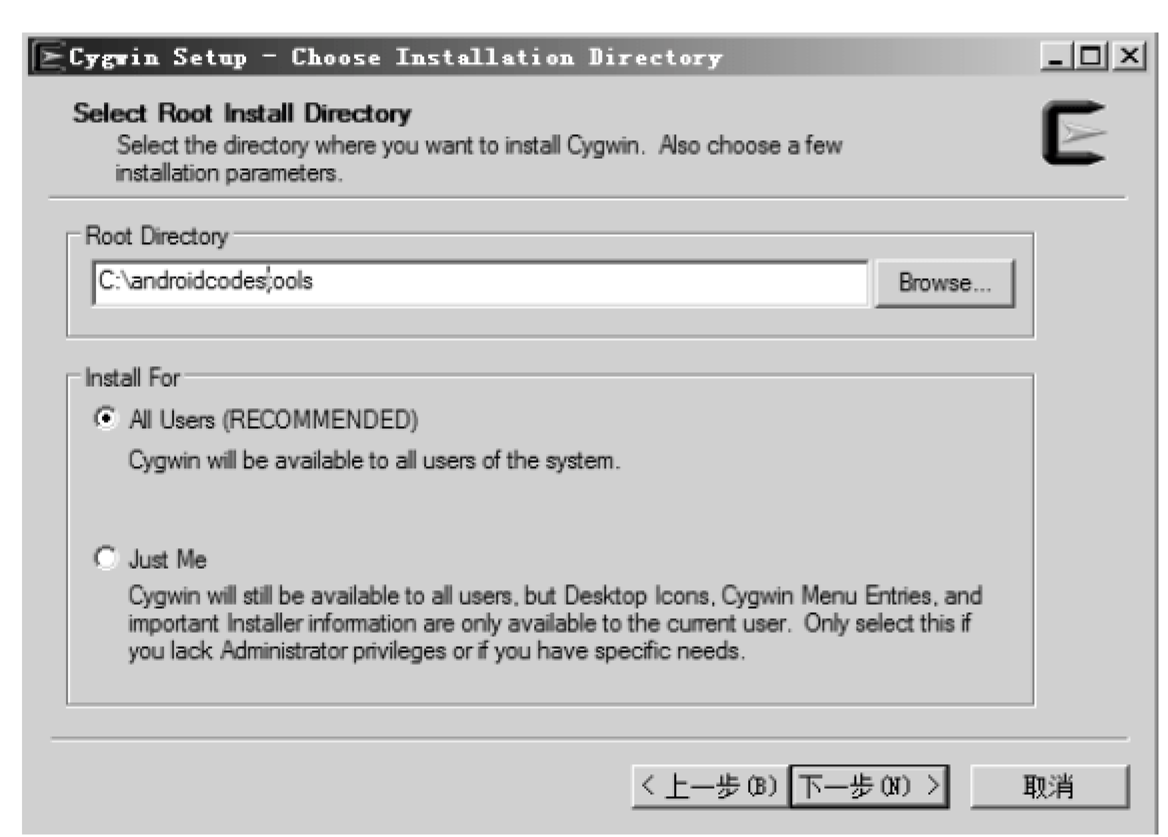


图 1-9　选择临时文件目录

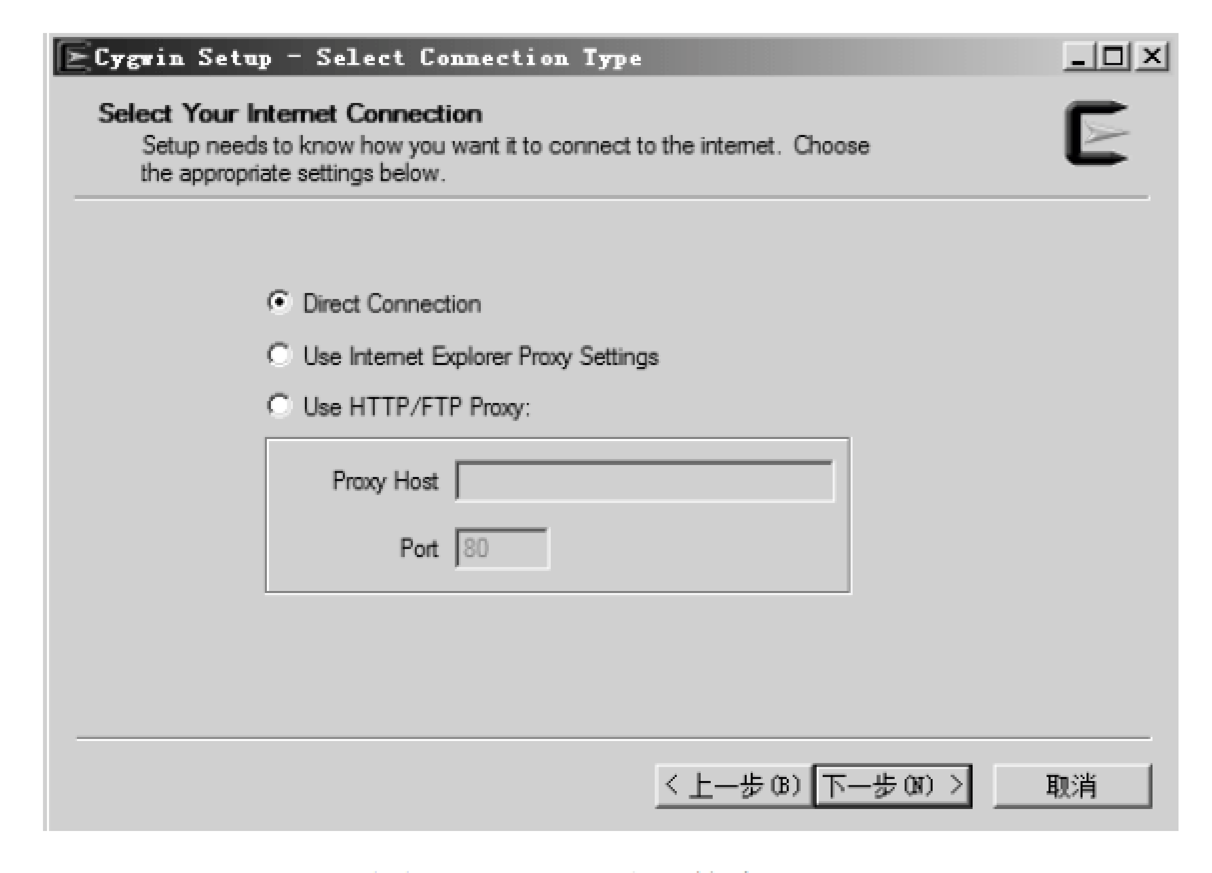


图 1-10　设置网络代理

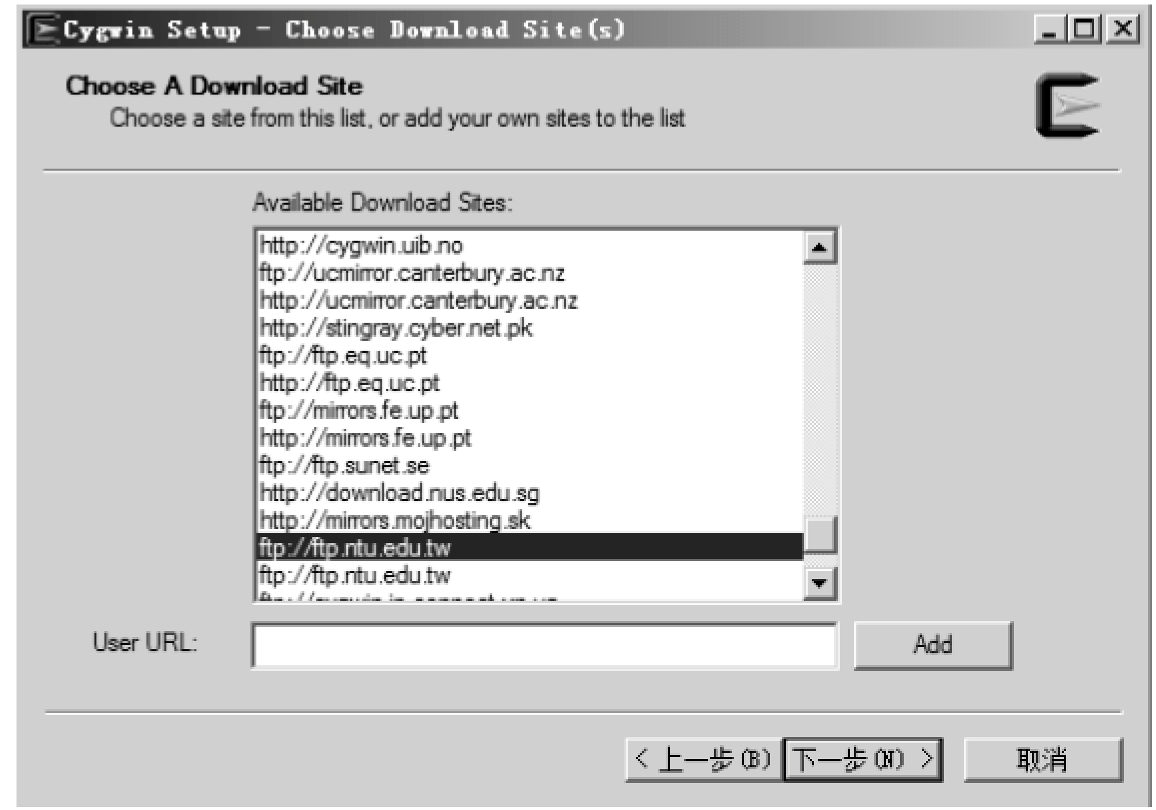


图 1-11　选择下载站点

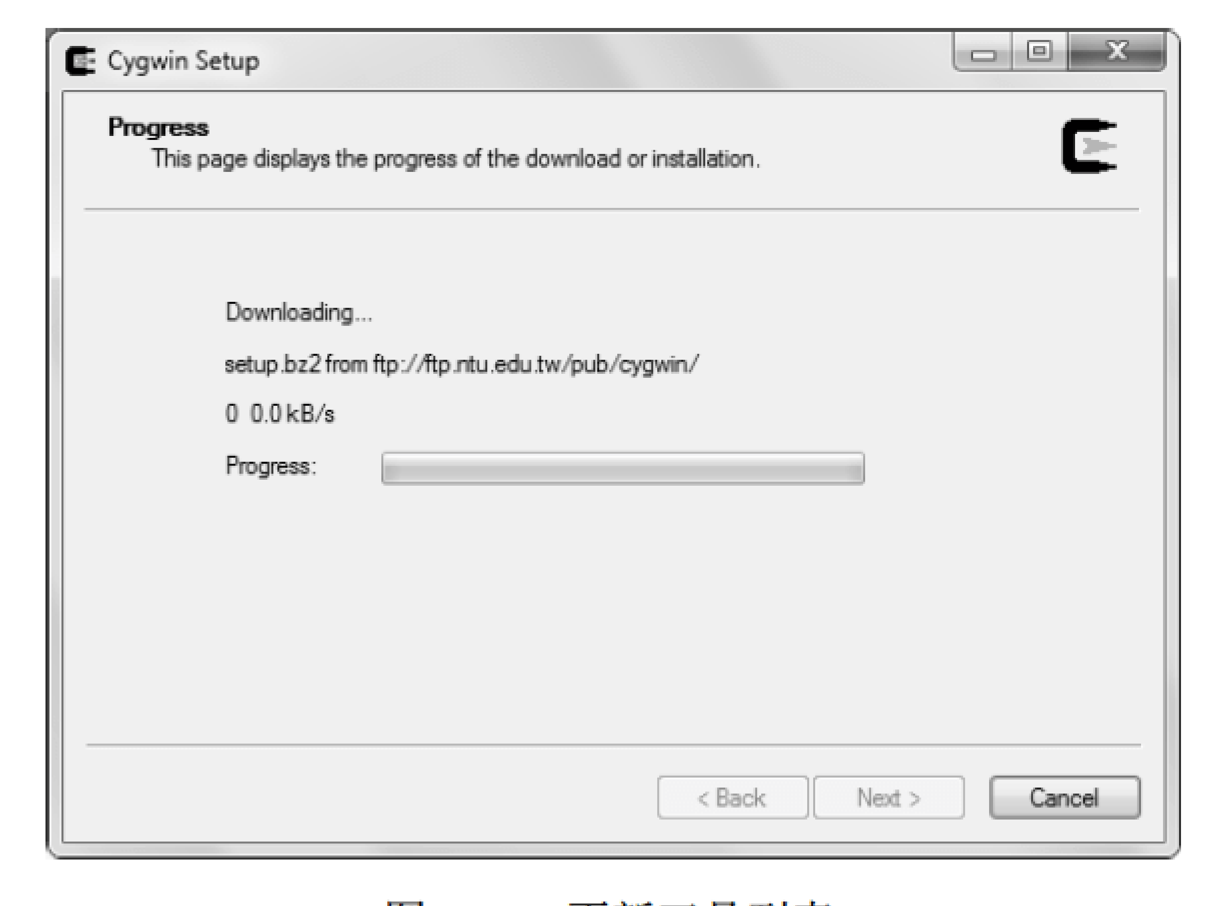


图 1-12　更新工具列表

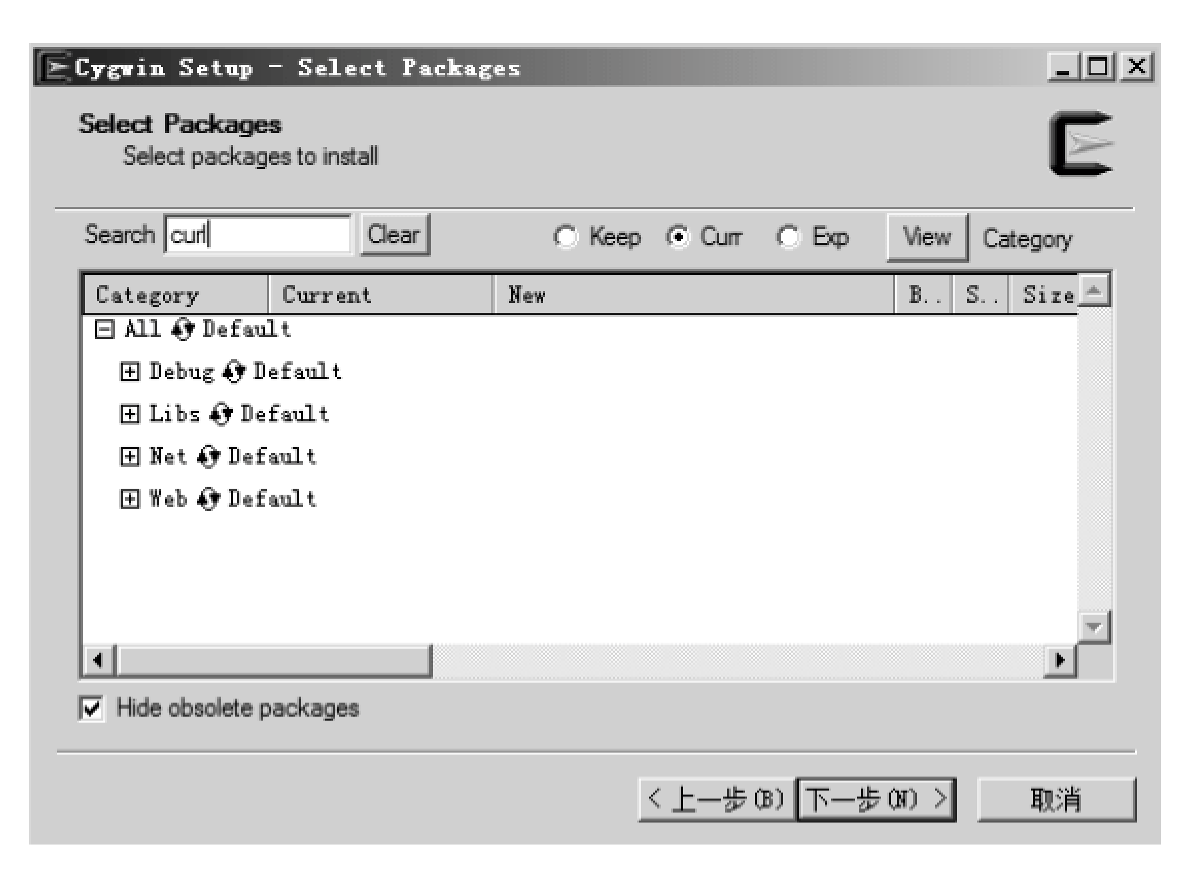


图 1-13　依次下载工具

为了确保能够安装上述工具，一定要用鼠标双击变为 Install 形式，如图 1-14 所示。

（9）单击“下一步”按钮，在弹出的对话框中进入漫长的等待过程，如图 1-15 所示。

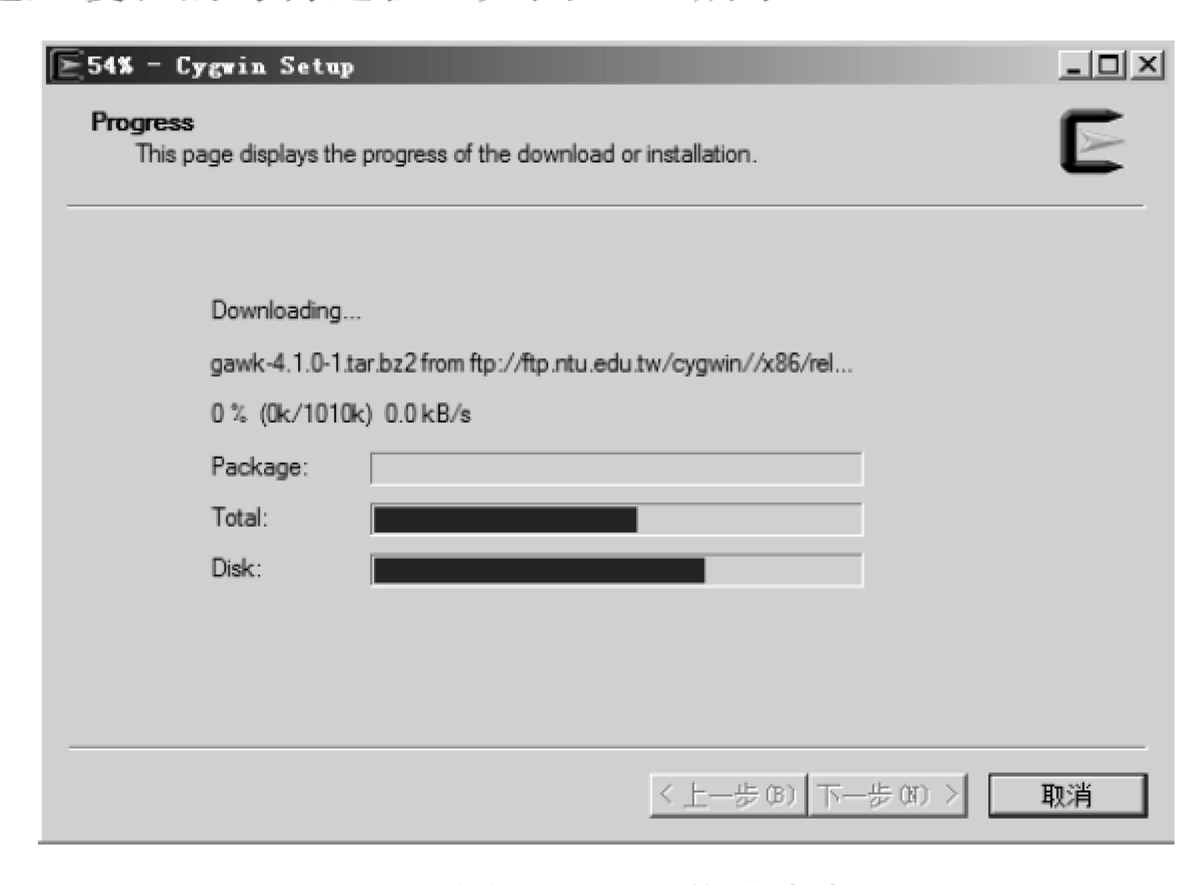


图 1-14　务必设置为 Install 形式　　　　图 1-15　下载进度条

如果下载安装成功会出现提示信息，单击“完成”按钮即完成安装。当安装好 cygwin 后，打开 cygwin，会模拟出一个 Linux 的工作环境，然后按照 Linux 平台的源码下载方法就可以下载 Android 源码了。

建议读者在下载 Android 源码时，严格按照官方提供的步骤进行，地址是 http://source.android.com/source/downloading.html，这一点对初学者来说尤为重要。另外，整个下载过程比较漫长，需要大家耐心等待。如图 1-16 所示是笔者机器的命令截图。

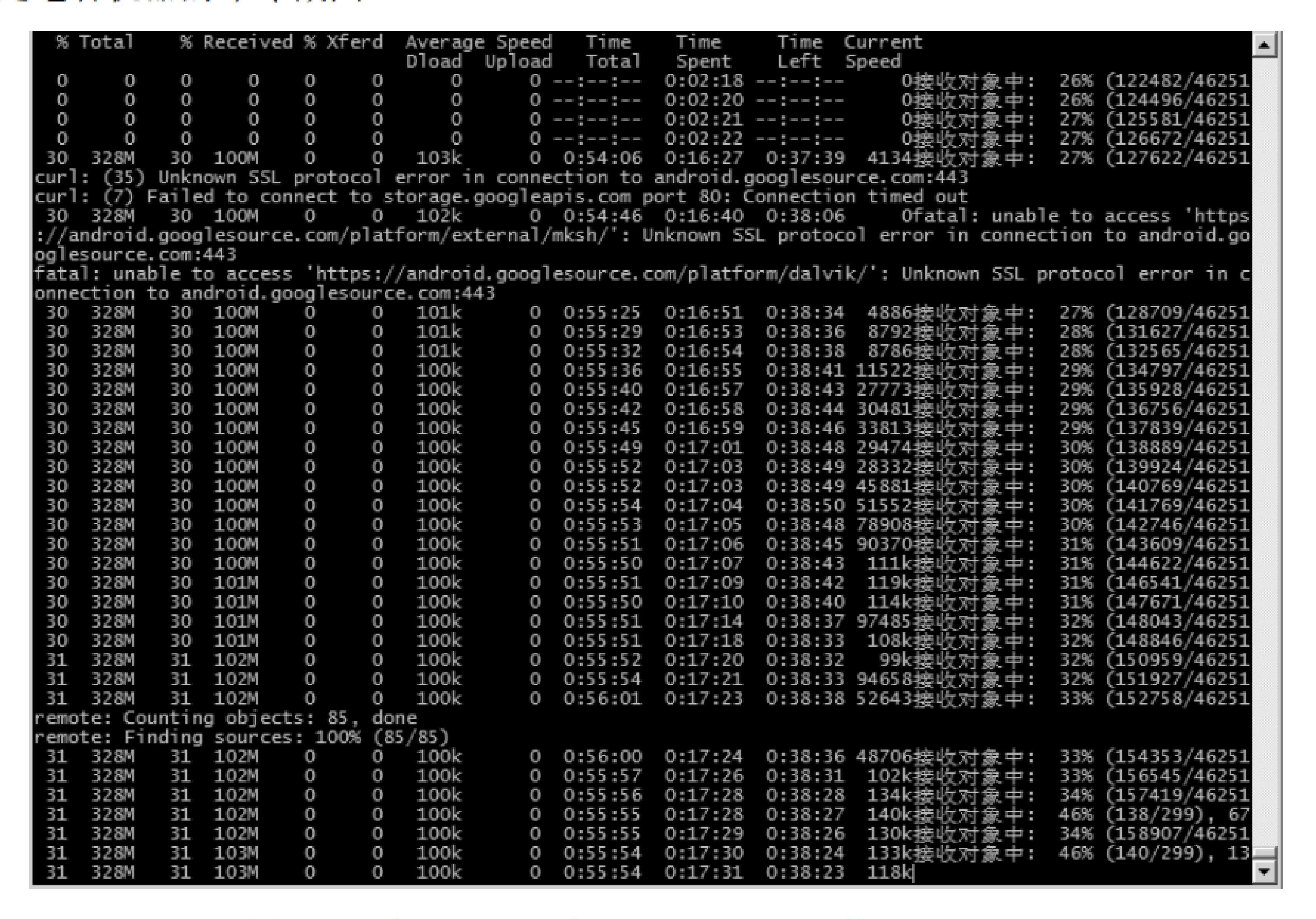


图 1-16　在 Windows 中用 cygwin 工具下载 Android 源码

1.4　分析 Android 源码结构

获得 Android 5.0 源码后，可以将源码的全部工程分为如下 3 个部分。

☑ Core Project：核心工程部分，这是建立 Android 系统的基础，被保存在根目录的各个文件夹中。

☑ External Project：扩展工程部分，可以使其他开源项目具有扩展功能，被保存在 external 文件夹中。

☑ Package：包部分，提供了 Android 的应用程序、内容提供者、输入法和服务，被保存在 package 文件夹中。

本节将详细讲解 Android 5.0 源码的目录结构。

1.4.1　总体结构

无论是 Android 1.5 还是 Android 5.0，各个版本的源码目录基本类似，里面包含了原始 Android 的目标机代码、主机编译工具和仿真环境。将下载的 Android 5.0 源码包解压缩后，可以看到第一级目录有多个文件夹和一个 Makefile 文件，如图 1-17 所示。

如果是编译后的源码目录，则会增加一个 out 文件夹，用来存放编译产生的文件。Android 5.0 第一级别目录结构的具体说明如表 1-1 所示。

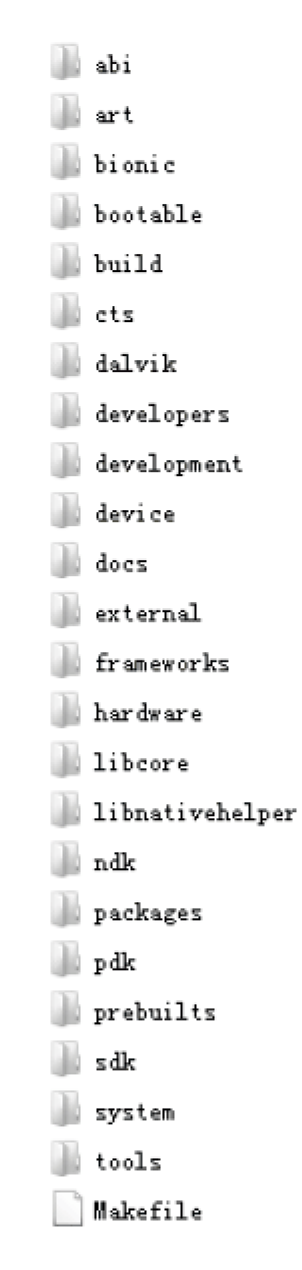


图 1-17　下载的 Android 5.0 内核

表 1-1　Android 5.0 源码的根目录

Android 源码根目录	描　述
abi	abi 相关代码，abi:application binary interface，应用程序二进制接口
art	全新的运行环境，需要和 Dalvik VM 区分开
bionic	bionic C 库
bootable	启动引导相关代码
build	存放系统编译规则及 generic 等基础开发配置包
cts	Android 兼容性测试套件标准
dalvik	dalvik Java 虚拟机
development	应用程序开发相关
device	设备相关代码
docs	介绍开源的相关文档
external	Android 使用的一些开源的模组
frameworks	核心框架——Java 及 C++语言，是 Android 应用程序的框架
gdk	即时通信模块
hardware	主要是硬件适配层 HAL 代码
kernel	Linux 的内核文件
libcore	核心库相关
libnativehelper	是 Support functions for Android's class libraries 的缩写，表示动态库，是实现 JNI 库的基础
ndk	ndk 相关代码。Android NDK（Android Native Development Kit）是一系列的开发工具，允许程序开发人员在 Android 应用程序中嵌入 C/C++语言编写的非托管代码
out	编译完成后的代码输出在此目录
packages	应用程序包

续表

Android 源码根目录	描　述
Pdk	Plug Development Kit 的缩写，是本地开发套件
prebuilts	x86 和 arm 架构下预编译的一些资源
sdk	sdk 及模拟器
system	文件系统和应用及组件，是用 C 语言实现的
tools	工具文件夹
vendor	厂商定制代码
Makefile	全局的 Makefile

由此可见，通过对源码中根目录的每个文件夹的功能介绍，可以看出源码按功能分类还是非常清晰的，可以分为系统代码、工具、文档、开发环境、虚拟机、配置脚本和编译脚本等类别，并且也可以看出涉及的内容比较庞大和复杂，源码分析工作需要多方面的理论和实践知识。

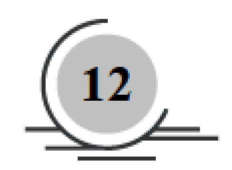
1.4.2　应用程序部分

应用程序主要是 UI 界面的实现，广大开发者基于 SDK 上开发的一个个独立的 APK 包，都是属于应用程序这一层的，应用程序在 Android 系统中处于最上层的位置。源码结构中的 packages 目录用来实现系统的应用程序，packages 的目录结构如下所示。

```
packages /
├─── apps                          //应用程序库
|    ├─── BasicSmsReceiver         //基础短信接收
|    ├─── Bluetooth                //蓝牙
|    ├─── Browser                  //浏览器
|    ├─── Calculator               //计算器
|    ├─── Calendar                 //日历
|    ├─── Camera                   //照相机
|    ├─── CellBroadcastReceiver    //单元广播接收
|    ├─── CertInstaller            //被调用的包，在 Android 中安装数字签名
|    ├─── Contacts                 //联系人
|    ├─── DeskClock                //桌面时钟
|    ├─── Email                    //电子邮件
|    ├─── Exchange                 //Exchange 服务
|    ├─── Gallery                  //图库
|    ├─── Gallery2                 //图库 2
|    ├─── HTMLViewer               //HTML 查看器
|    ├─── KeyChain                 //密码管理
|    ├─── Launcher2                //启动器 2
|    ├─── Mms                      //彩信
|    ├─── Music                    //音乐
|    ├─── MusicFX                  //音频增强
|    ├─── Nfc                      //近场通信
|    ├─── PackageInstaller         //包安装器
|    ├─── Phone                    //电话
|    ├─── Protips                  //主屏幕提示
|    ├─── Provision                //引导设置
|    ├─── QuickSearchBox           //快速搜索框
```

```
|       ├──── Settings                    //设置
|       ├──── SoundRecorder               //录音机
|       ├──── SpareParts                  //系统设置
|       ├──── SpeechRecorder              //录音程序
|       ├──── Stk                         //SIM 卡相关
|       ├──── Tag                         //标签
|       ├──── VideoEditor                 //视频编辑
|       └──── VoiceDialer                 //语音编号
├──── experimental                        //非官方的应用程序
|       ├──── BugReportSender             //bug 的报告程序
|       ├──── Bummer
|       ├──── CameraPreviewTest           //照相机预览测试程序
|       ├──── DreamTheater
|       ├──── ExampleImsFramework
|       ├──── LoaderApp
|       ├──── NotificationLog
|       ├──── NotificationShowcase
|       ├──── procstatlog
|       ├──── RpcPerformance
|       └──── StrictModeTest
├──── inputmethods                        //输入法
|       ├──── LatinIME                    //拉丁文输入法
|       ├──── OpenWnn                     //OpenWnn 输入法
|       └──── PinyinIME                   //拼音输入法
├──── providers                           //提供器
|       ├──── ApplicationsProvider        //应用程序提供器，提供应用程序所需的界面
|       ├──── CalendarProvider            //日历提供器
|       ├──── ContactsProvider            //联系人提供器
|       ├──── DownloadProvider            //下载管理提供器
|       ├──── DrmProvider                 //数据库相关
|       ├──── GoogleContactsProvider      //Google 联系人提供器
|       ├──── MediaProvider               //媒体提供器
|       ├──── TelephonyProvider           //彩信提供器
|       └──── UserDictionaryProvider      //用户字典提供器
├──── screensavers                        //屏幕保护
|       ├──── Basic                       //基本屏幕保护
|       ├──── PhotoTable                  //照片方格
|       ├──── WebView                     //网页
└──── wallpapers                          //墙纸
      ├──── Basic                         //系统内置墙纸
      ├──── Galaxy4                       //S4 内置墙纸
      ├──── HoloSpiral                    //手枪皮套墙纸
      ├──── LivePicker
      ├──── MagicSmoke
      ├──── MusicVisualization
      ├──── NoiseField
      └──── PhaseBeam
```

通过上面的目录结构可以看出，package 目录主要存放的是 Android 系统应用层相关的内容，包括应用程序相关的包或者资源文件，其中即包括系统自带的应用程序，又有第三方开发的应用程序，还有屏幕保护和墙纸等应用，所以源码中 package 目录对应着系统的应用层。



1.4.3　应用程序框架部分

应用程序框架是 Android 系统中的核心部分，也就是 SDK 部分，它会提供接口给应用程序使用，同时应用程序框架又会和系统服务、系统程序库、硬件抽象层有关联，所以其作用十分重大，应用程序框架的实现代码大部分都在/frameworks/base 和/frameworks/av 目录下，/frameworks/base 的目录结构如下所示。

```
frameworks/base
├── api                    //全是 XML 文件，定义了 API
├── cmds                   //Android 中的重要命令（am、app_proce 等）
├── core                   //核心库
├── data                   //声音字体等数据文件
├── docs                   //文档
├── drm                    //数字版权管理
├── graphics               //图形图像
├── icu4j                  //用于解决国际化问题
├── include                //头文件
├── keystore               //数字签名证书相关
├── libs                   //库
├── location               //地理位置
├── media                  //多媒体
├── native                 //本地库
├── nfc-extras             //NFC 相关
├── obex                   //蓝牙传输
├── opengl                 //opengl 相关
├── packages               //设置，TTS，VPN 程序
├── policy                 //锁屏界面相关
├── sax                    //XML 解析器
├── services               //Android 的服务
├── telephony              //电话相关
├── test-runner            //测试相关
├── tests                  //测试相关
├── tools                  //工具
├── voip                   //可视通话
└── wifi                   //无线网络
```

以上这些文件夹包含了应用程序框架层的大部分代码，正是这些目录下的文件构成了 Android 的应用程序框架层，暴露出接口给应用程序调用，同时衔接系统程序库和硬件抽象层，形成一个由上至下的调用过程。在/frameworks/base 目录下也涉及系统服务和程序库中的一些代码，后面的两个小节中将会详细分析。

1.4.4　系统服务部分

在 1.4.3 节中介绍了应用程序框架层的内容，了解到大部分的实现代码保存在/frameworks/base 目录下。其实在这个目录中还有一个名为 services 的目录，里面的代码是用于实现 Android 系统服务的。接下来将详细介绍 services 目录下的内容，其目录结构如下所示。

```
frameworks/base/services
├── common_time            //时间日期相关的服务
├── input                  //输入系统服务
├── java                   //其他重要服务的 Java 层
├── jni                    //其他重要服务的 JNI 层
└── tests                  //测试相关
```

其中 java 和 jni 两个目录分别是一些其他的服务的 Java 层和 JNI 层实现，java 目录下更详细的目录结构以及其他 Android 系统服务的说明如下所示。

```
frameworks/base/services/java/com/android/server
├── accessibility
├── am
├── connectivity
├── display
├── dreams
├── drm
├── input
├── location
├── net
├── pm
├── power
├── updates
├── usb
├── wm
├── AlarmManagerService.java                    //闹钟服务
├── AppWidgetService.java                       //应用程序小工具服务
├── AppWidgetServiceImpl.java
├── AttributeCache.java
├── BackupManagerService.java                   //备份服务
├── BatteryService.java                         //电池相关服务
├── BluetoothManagerService.java                //蓝牙
├── BootReceiver.java
├── BrickReceiver.java
├── CertBlacklister.java
├── ClipboardService.java
├── CommonTimeManagementService.java            //时间管理服务
├── ConnectivityService.java
├── CountryDetectorService.java
├── DevicePolicyManagerService.java
├── DeviceStorageMonitorService.java            //设备存储器监听服务
├── DiskStatsService.java                       //磁盘状态服务
├── DockObserver.java                           //底座监视服务
├── DropBoxManagerService.java
├── EntropyMixer.java
├── EventLogTags.logtags
├── INativeDaemonConnectorCallbacks.java
├── InputMethodManagerService.java              //输入法管理服务
├── IntentResolver.java
├── IntentResolverOld.java
├── LightsService.java
├── LocationManagerService.java                 //地理位置服务
├── MasterClearReceiver.java
├── MountService.java                           //挂载服务
├── NativeDaemonConnector.java
├── NativeDaemonConnectorException.java
├── NativeDaemonEvent.java
├── NetworkManagementService.java               //网络管理服务
```

```
├──── NetworkTimeUpdateService.java
├──── NotificationManagerService.java          //通知服务
├──── NsdService.java
├──── PackageManagerBackupAgent.java
├──── PreferredComponent.java
├──── ProcessMap.java
├──── RandomBlock.java
├──── RecognitionManagerService.java
├──── SamplingProfilerService.java
├──── SerialService.java                       //NFC 相关
├──── ServiceWatcher.java
├──── ShutdownActivity.java
├──── StatusBarManagerService.java             //状态栏管理服务
├──── SystemBackupAgent.java
├──── SystemServer.java
├──── TelephonyRegistry.java
├──── TextServicesManagerService.java
├──── ThrottleService.java
├──── TwilightCalculator.java
├──── TwilightService.java
├──── UiModeManagerService.java
├──── UpdateLockService.java                   //锁屏更新服务
├──── VibratorService.java                     //振动服务
├──── WallpaperManagerService.java             //壁纸服务
├──── Watchdog.java                            //看门狗
├──── WifiService.java                         //无线网络服务
└──── WiredAccessoryManager.java               //无线设备管理服务
```

从上面的文件夹和文件可以看出，Android 中涉及的服务种类非常多，包括界面、网络、电话等核心模块基本上都有其专属的服务，这些是属于系统级别的服务，这些系统服务一般都会在 Android 系统启动时加载，在系统关闭时结束，受到系统的管理，应用程序并没有权力去打开或者关闭，它们会随着系统的运行一直在后台运行，供应用程序和其他的组件来使用。

另外在 frameworks/av/下面也有一个 services 目录，这个目录下存放的是音频和照相机的服务的实现代码，目录结构如下所示。

```
frameworks/av/services
├──── audioflinger                             //音频管理服务
└──── camera                                   //照相机的管理服务
```

这个 av/services 目录下的文件主要是用来支持 Android 系统中的音频和照相机服务的，这是两个非常重要的系统服务，开发应用程序时会经常依赖这两个服务。

1.4.5　系统程序库部分

Android 的系统程序库类型非常多，功能也非常强大，正是有了这些程序库，Android 系统才能运行多种多样的应用程序。在接下来的内容中，笔者挑了一些很常用也很重要的系统程序库来分析它们在源码中所处的位置。

（1）系统 C 库

Android 系统采用的是一个从 BSD 继承而来的标准的系统函数库 bionic，在源码根目录下有这个文件夹，其目录结构如下所示。

```
bionic/
├── libc                          //C 库
├── libdl                         //动态链接库相关
├── libm                          //数学库
├── libstdc++                     //C++实现库
├── libthread_db                  //线程库
├── linker                        //连接器相关
└── test                          //测试相关
```

（2）媒体库

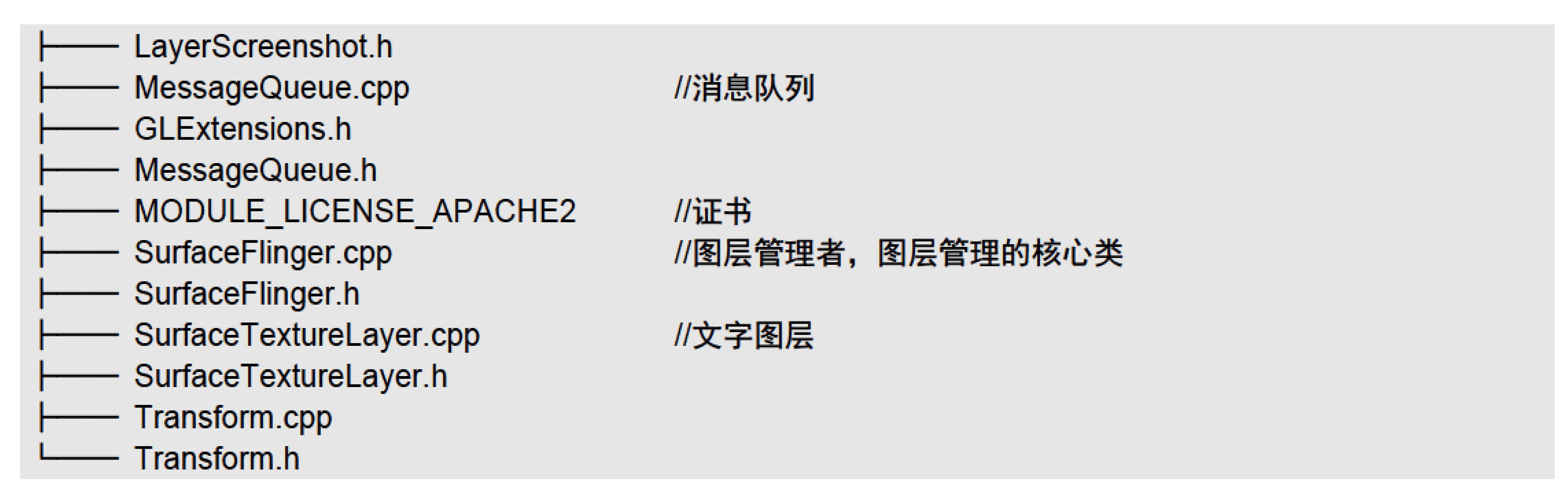
Anroid 中的媒体库在 2.3 之前是由 OpenCore 实现的，2.3 之后 Stragefright 被替换了，OpenCore 成为了新的多媒体的实现库。同时 Android 也自带了一些音视频的管理库，用于管理多媒体的录制、播放、编码和解码等功能。Android 的多媒体程序库的实现代码主要在/frameworks/av/media 目录下，其目录结构如下所示。

```
frameworks/av/media/
├── common_time                   //时间相关
├── libeffects                    //多媒体效果
├── libmedia                      //多媒体录制，播放
├── libmedia_native               //里面只有一个 Android.mk，用来编译 native 文件
├── libmediaplayerservice         //多媒体播放服务的实现库
├── libstagefright                //stagefright 的实现库
├── mediaserver                   //跨进程多媒体服务
└── mtp                           //MTP 协议的实现（媒体传输协议）
```

（3）图层显示库

Android 中的图层显示库主要负责对显示子系统的管理，负责图层的渲染、叠加、绘制等功能，提供了 2D 和 3D 图层的无缝融合，是整个 Android 系统显示的“大脑中枢”，其代码在/frameworks/native/services/surfaceflinger/目录下，其目录结构如下所示。

```
frameworks/native/services/surfaceflinger/
├── DisplayHardware               //显示底层相关
├── tests                         //测试
├── Android.mk                    //MakeFile 文件
├── Barrier.h
├── Client.cpp                    //显示的客户端实现文件
├── Client.h
├── clz.cpp
├── clz.h
├── DdmConnection.cpp
├── DdmConnection.h
├── DisplayDevice.cpp             //显示设备相关
├── DisplayDevice.h
├── EventThread.cpp               //消息线程
├── EventThread.h
├── GLExtensions.cpp              //Opengl 扩展
├── GLExtensions.h
├── Layer.cpp                     //图层相关
├── Layer.h
├── LayerBase.cpp                 //图层基类
├── LayerBase.h
├── LayerDim.cpp                  //图层相关
├── LayerDim.h
├── LayerScreenshot.cpp           //图层相关
```



```
├── LayerScreenshot.h
├── MessageQueue.cpp                          //消息队列
├── GLExtensions.h
├── MessageQueue.h
├── MODULE_LICENSE_APACHE2                    //证书
├── SurfaceFlinger.cpp                        //图层管理者，图层管理的核心类
├── SurfaceFlinger.h
├── SurfaceTextureLayer.cpp                   //文字图层
├── SurfaceTextureLayer.h
├── Transform.cpp
└── Transform.h
```

（4）网络引擎库

网络引擎库主要是用来实现 Web 浏览器的引擎，支持 Android 的 Web 浏览器和一个可嵌入的 Web 视图，这个是采用第三方开发的浏览器引擎 webkit 实现的，webkit 的代码在/external/webkit/目录下，其目录结构如下所示。

```
external/webkit/
├── Examples                                  //webkit 例子
├── LayoutTests                               //布局测试
├── PerformanceTests                          //表现测试
├── Source                                    //webkit 源代码
├── Tools                                     //工具
├── WebKitLibraries                           //webkit 用到的库
├── Android.mk                                //Makefile
├── bison_check.mk
├── CleanSpec.mk
├── MODULE_LICENSE_LGPL                       //证书
├── NOTICE
└── WEBKIT_MERGE_REVISION                     //版本信息
```

（5）3D 图形库

Android 中的 3D 图形渲染是采用 Opengl 来实现的，Opengl 是开源的第三方图形渲染库，使用该库可以实现 Android 中的 3D 图形硬件加速或者 3D 图形软件加速功能，是一个非常重要的功能库。从 Android 5.0 开始，支持最新最强大的 OpenGL ES 3.1。其实现代码在/frameworks/native/opengl 中，目录结构如下所示。

```
frameworks/native/opengl/
├── include                                   //Opengl 中的头文件
├── libagl                                    //在 mac os 上的库
├── libs                                      //Opengl 的接口和实现库
├── specs                                     //Opengl 的文档
├── tests                                     //测试相关
└── tools                                     //工具库
```

（6）SQLite

SQLite 是 Android 系统自带的一个轻量级关系数据库，其实现源代码已经在网上开源。SQLite 的优点是操作简单方便，运行速度较快，占用资源较少，比较适合在嵌入式设备上使用。SQLite 是 Android 系统自带的实现数据库功能的核心库，其代码实现分为 Java 和 C 两个部分，Java 部分的代码在/frameworks/base/core/java/android/database，目录结构如下所示。

```
frameworks/base/core/java/android/database/
├── sqlite                                    //SQLite 的框架文件
├── AbstractCursor.java                       //游标的抽象类
```

```
├── AbstractWindowedCursor.java
├── BulkCursorDescriptor.java
├── BulkCursorNative.java
├── BulkCursorToCursorAdaptor.java              //游标适配器
├── CharArrayBuffer.java
├── ContentObservable.java
├── ContentObserver.java                        //内容观察者
├── CrossProcessCursor.java
├── CrossProcessCursorWrapper.java              //CrossProcessCursor 的封装类
├── Cursor.java                                 //游标实现类
├── CursorIndexOutOfBoundsException.java        //游标出界异常
├── CursorJoiner.java
├── CursorToBulkCursorAdaptor.java              //适配器
├── CursorWindow.java                           //游标窗口
├── CursorWindowAllocationException.java        //游标窗口异常
├── CursorWrapper.java                          //游标封装类
├── DatabaseErrorHandler.java                   //数据库错误句柄
├── DatabaseUtils.java                          //数据库工具类
├── DataSetObservable.java
├── DataSetObserver.java
├── DefaultDatabaseErrorHandler.java            //默认数据库错误句柄
├── IBulkCursor.java
├── IContentObserver.aidl                       //aidl 用于跨进程通信
├── MatrixCursor.java
├── MergeCursor.java
├── Observable.java
├── package.html
├── SQLException.java                           //数据库异常
└── StaleDataException.java
```

Java 层的代码主要是实现 SQLite 的框架和接口的实现，方便用户开发应用程序时能简单地操作数据库，并且捕获数据库异常。

C++层的代码在/external/sqlite 路径下，其目录结构如下所示。

```
external/sqlite/
├── android                                     //Android 数据库的一些工具包
└── dist                                        //Android 数据库底层实现
```

从上面 Java 和 C 部分的代码目录结构可以看出，SQLite 在 Android 中还是有很重要的地位的，并且在 SDK 中会有开放的接口让应用程序可以很简单方便地操作数据库，对数据进行存储和删除。

1.4.6 系统运行库部分

众所周知，Android 系统的应用层是采用 Java 开发的，由于 Java 语言的跨平台特性，Java 代码必须运行在虚拟机中。正是因为这个特性，Android 系统也自己实现了一个类似 JVM 但是更适用于嵌入式平台的 Java 虚拟机，被称为 Dalvik。

Dalvik 功能等同于 JVM，为 Android 平台上的 Java 代码提供了运行环境，Dalvik 本身是由 C++语言实现的，在源码中根目录下有 dalvik 文件夹，里面存放的是 Dalvik 虚拟机的实现代码，其目录结构如下所示。

```
./
├── dalvikvm                                    //入口目录
├── dexdump                                     //dex 反汇编
```

```
├─── dexgen                    //dex 生成相关
├─── dexlist                   //dex 列表
├─── dexopt                    //与验证和优化
├─── docs                      //文档
├─── dvz                       //zygot 相关
├─── dx                        //dx 工具，将多个 Java 转换为 dex
├─── hit
├─── libdex                    //dex 库的实现代码
├─── opcode-gen
├─── tests                     //测试相关
├─── tools                     //工具
├─── unit-tests                //测试相关
├─── vm                        //虚拟机的实现
├─── Android.mk                //Makefile
├─── CleanSpec.mk
├─── MODULE_LICENSE_APACHE2
├─── NOTICE
└─── README.txt
```

正是有上面这些代码实现的 Android 虚拟机，所以应用程序生成的二进制执行文件能够快速、稳定地运行在 Android 系统上。

而从 Android 5.0 开始，Android 应用程序的默认运行环境为 ART，ART 模式拥有更快更高的运行效率，其目录结构如图 1-18 所示。



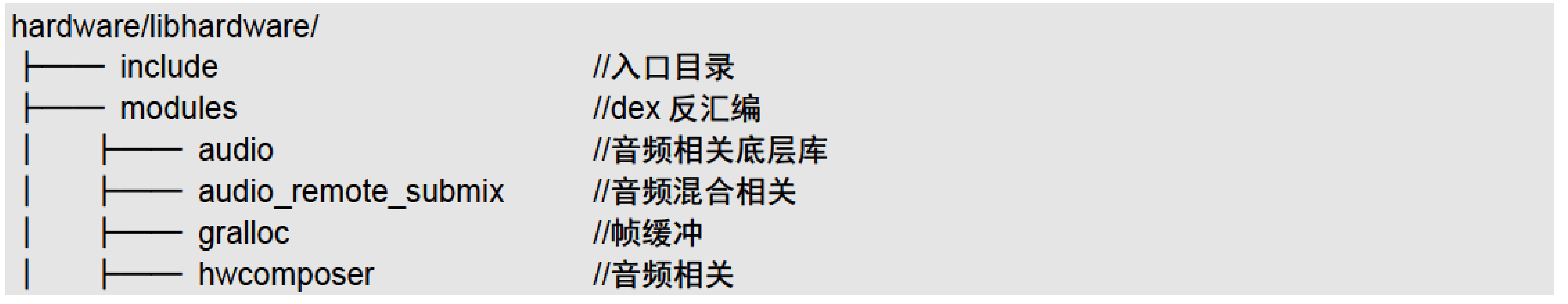
图 1-18　ART 模块的目录结构

1.4.7　硬件抽象层部分

Android 的硬件抽象是各种功能的底层实现，理论上不同的硬件平台会有不同的硬件抽象层实现，这一个层次也是和驱动层、硬件层有紧密联系的，起着承上启下的作用，对上要实现应用程序框架层的接口，对下要实现一些硬件基本功能，以及调用驱动层的接口。需要注意的是，这一层也是广大 OEM 厂商改动最大的一层，因为这一层的代码和终端采用什么样硬件的硬件平台有很大关系。源码中存放的是硬件抽象层框架的实现代码和一些平台无关性的接口的实现。硬件抽象层代码在源码根目录下的 hardware 文件夹中，其目录结构如下所示。

```
hardware/
├─── libhardware               //新机制硬件库
├─── libhardware_legacy        //旧机制硬件库
└─── ril                       //ril 模块相关的底层实现
```

从上面的目录结构可以看出，硬件抽象层中主要是实现了一些底层的硬件库，用来实现应用层框架层中的功能，具体硬件库中有哪些内容，我们可以继续细分其目录结构，例如 libhardware 目录下的结构如下所示。



```
hardware/libhardware/
├─── include                   //入口目录
├─── modules                   //dex 反汇编
|    ├─── audio                //音频相关底层库
|    ├─── audio_remote_submix  //音频混合相关
|    ├─── gralloc              //帧缓冲
|    ├─── hwcomposer           //音频相关
```

```
|       ├─── local_time                    //本地时间
|       ├─── nfc                           //NFC 功能
|       ├─── nfc-nci                       //NFC 的接口
|       ├─── power                         //电源
|       ├─── usbaudio                      //USB 音频设备
|       ├─── Android.mk                    //Makefile
|       ├─── README.android
├─── tests                                 //dex 生成相关
├─── dexlist                               //dex 列表
├─── dexopt                                //与验证和优化
└─── docs                                  //文档
```

从上面的目录结构可以分析出，libhardware 目录主要是 Android 系统的某些功能的底层实现，包括 audio、nfc、power。

而 libhardware_legacy 的目录与 libhardware 大同小异，只是针对旧的实现方式做的一套硬件库，其目录下还有 uevent、wifi 以及虚拟机的底层实现。这两个目录下的代码一般会由设备厂家根据自身的硬件平台来实现符合 Android 机制的硬件库。

而 ril 目录下存放的是无线硬件设备与电话的实现，其目录结构如下所示。

```
hardware/ril/
├─── include                               //头文件
├─── libril                                //libril 库
├─── mock-ril
├─── reference-ril                         //reference ril 库
├─── rild                                  //ril 守护进程
└─── CleanSpec.mk
```

1.5 编译源码

在进行 Android 底层驱动开发工作之前，需要先编译获取的开源代码。编译方法是使用 Android 源码根目录下的 Makefile，并执行 make 命令来实现。当然在编译 Android 源码之前，首先要确定已经完成同步工作。进入 Android 源码目录使用 make 命令进行编译，使用此命令的格式如下所示。

```
$: cd ~/Android5.0（这里的 Android5.0 就是我们下载源码的保存目录）
$: make
```

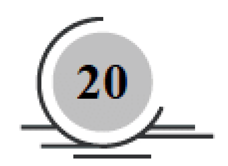
本节将详细讲解编译并在模拟器中运行 Android 5.0 源码的方法。

1.5.1 搭建编译环境

在编译 Android 源码之前，需要先进行环境搭建工作。下面以 Ubuntu 系统为例讲解搭建编译环境以及编译 Android 源码的方法，具体流程如下。

（1）安装 JDK，编译 Android 5.0 的源码需要 JDK 1.7，下载 jdk-7u21-linux-i586.bin 后进行安装，对应命令如下。

```
$ cd /usr
$ mkdir java
$ cd java
$ sudo cp jdk-7u21-linux-i586.bin 所在目录 ./
```



```
$ sudo chmod 755 jdk-7u21-linux-i586.bin
$ sudo sh jdk-7u21-linux-i586.bin
```

（2）设置 JDK 环境变量，将如下环境变量添加到主文件夹目录下的.bashrc 文件中，然后用 source 命令使其生效，加入的环境变量代码如下。

```
export JAVA_HOME=/usr/java/jdk1.7.0_27
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$PATH:$JAVA_HOME/bin:$JAVA_HOME/bin/tools.jar:$JRE_HOME/bin
export ANDROID_JAVA_HOME=$JAVA_HOME
```

对于安装好的 JDK，并且在添加环境变量之后，可以输入并执行命令 java-version 来查看 JDK 的版本。

（3）安装需要的编译工具，对于 Linux 10.04 系统来说，只需要安装如下所示的软件工具即可，在安装前保持电脑正常连接网络。

```
sudo apt-get install git-core gnupg flex bison gperf build-essential \
zip curl zlib1g-dev libc6-dev lib32ncurses5-dev ia31-libs \
x11proto-core-dev libx11-dev lib32readline5-dev lib32z-dev \
libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown \
```

然后使用下面的命令做一个软链接文件。

```
sudo ln -s /usr/lib32/mesa/libGL.so.1 /usr/lib32/mesa/libGL.so
```

然后安装对于 11.10 系统需要的特别工具。

```
sudo apt-get install libx11-dev:i386
```

（4）开始设置高速缓存，目的是加快编译速度。对于配置不是很高的电脑来说，最好进行这个设置，这样可以节约很多时间。设置方法是先用 vi 或者 gedit 软件打开宿主目录下的“.bashrc”文件，然后在文件的最后添加如下值。

```
export USE_CCACHE=1
```

然后保存后退出，重新登录系统以使设置生效，如图 1-19 所示。

```
File Edit View Terminal Help
fi

# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
    . /etc/bash_completion
fi
export USE_CCACHE=1
                                        100,19        Bot
```

图 1-19 设置高速缓存

在终端中切换到源码根目录中，然后执行下面的命令设置 ccache 的大小为 50GB。

```
prebuilts/misc/linux-x86/ccache/ccache -M 50G
```

其实 ccache 就是一个执行文件，后面的-M 和 50G 是传递给 ccache 的参数，表示设置 50GB 的缓存空间，这个大小可以根据我们的时间需要来修改。

（5）运行如下命令，导入编译 Android 源码所需的环境变量和其他参数。

```
source build/envsetup.sh
```

（6）运行 lunch 命令选择编译目标，运行 lunch 命令后会出现一些已经预置好的项目。在此输入对应的数字，然后回车选择编译目标对象。

（7）运行 lunch 命令并选择好编译目标，运行如下命令进行编译。

```
make -j16
```

编译过程比较慢，因为电脑配置的问题可能需要几个小时的漫长等待。编译成功后会弹出如图 1-20 所示的提示信息。

```
File  Edit  View  Terminal  Help
+ MAKE_EXT4FS_CMD='make_ext4fs  -S out/target/product/generic/root/file_contexts -l 576716800 -a
system out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/p
roduct/generic/system'
+ echo make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/ta
rget/product/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/product/generi
c/system
make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/target/pr
oduct/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/product/generic/syste
m
+ make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/target/
product/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/product/generic/sys
tem
Creating filesystem with parameters:
    Size: 576716800
    Block size: 4096
    Blocks per group: 32768
    Inodes per group: 7040
    Inode size: 256
    Journal blocks: 2200
    Label:
    Blocks: 140800
    Block groups: 5
    Reserved block group size: 39
Created filesystem with 1262/35200 inodes and 81850/140800 blocks
+ '[' 0 -ne 0 ']'
Install system fs image: out/target/product/generic/system.img
out/target/product/generic/system.img+ maxsize=588791808 blocksize=2112 total=576716800 reserve=5
947392
```

图 1-20　编译成功时的提示信息

这样在编译完成后，可以在源码中的 out/target/product/generic/目录生成对应固件等文件，如图 1-21 所示。



图 1-21　out/target/product/generic/目录

注意： 获取 Android 源码的过程是一个漫长的过程，一个疏忽大意就可能造成下载失败的结果。另外，编译 Android 源码的过程也是一个需要耐心的过程，笔者在第一次编译过程中也走了不少弯路。幸亏网络中有很多网友发的教程帖子，例如笔者就参考了网名为 xyh666168 的帖子，地址是 http://jingyan.baidu.com/article/a501d80ce61ad0ec630f5e0b.html。因为大家的机器配置是各种各样的，CPU 的型号参数也不同，所以建议读者多参考网络中的教程和解决方案。

1.5.2　在模拟器中运行

最后在模拟器中运行的步骤就比较简单了，只需在终端中执行下面的命令即可。

```
emulator
```

运行成功后的效果如图 1-22 所示。

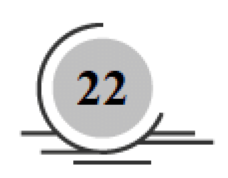




图 1-22　在模拟器中的编译执行效果

1.5.3　编译源码生成 SDK

平时大部分 Android 应用程序开发是基于 SDK 实现的，其过程是使用 SDK 中的接口实现各种各样的功能。我们可以在 Android 的官方网站上直接下载最新的 SDK 版本，也可以从源码中生成 SDK，因为源码中也包含有 SDK 的代码。

我们下载的 Android 5.0 的源码的根目录下有一个 SDK 目录，所有的 SDK 相关的代码都放在这个目录下，包括镜像文件、模拟器、ADB 等常用工具以及 SDK 中的开发包的文档，我们可以通过编译的方式来生成开发需要的 SDK，编译命令如下所示。

```
$ Make SDK
```

当编译完成后，会在/out/host/linux-x86/sdk/目录下生成 SDK，这个 SDK 完全和源码同步，与官网上下载的 SDK 功能完全相同，会有开发用的 JAR 包、模拟器管理工具、ADB 调试工具，可以使用这个编译生成的 SDK 来开发我们的应用程序。

对于 Android 系统的开发，基本可以分为如下两种开发方式。

☑　基于 SDK 的开发。

☑　基于源码的开发。

在一般情况下，开发的应用程序都是基于 SDK 的开发，比较方便而且兼容性比较好。基于源码的开发相对于基于 SDK 的开发要求对源码的架构认识更深刻，一般用于需要修改系统层面的场合。两种方式应用场景不同，各有优缺点，本节将主要介绍基于 SDK 的开发。

如果想基于 SDK 开发 Android 的应用程序，我们需要 JDK、SDK 和一个开发环境，JDK 和 SDK 在不同的平台下有不同的版本，本章主要讨论 Windows 7 平台下的开发环境搭建。

（1）安装 JDK

由于 Android 的应用程序是使用 Java 语言开发的，所以首先需要安装 Java 的 JDK，下载链接：http://java.sun.com/javase/downloads/index.jsp，进入后选择合适的平台以及下载最新版本的 JDK，安装成功后命令行下可以查看 JDK 版本。

（2）安装 Eclipse

Eclipse 是开发 Android 应用程序的 IDE 环境，有非常丰富的插件可以使用，单击 http://www.eclipse.

org/downloads/可以下载合适平台的最新版本 Eclipse。

（3）安装 Android SDK

Android SDK 是 Google 对外发布的专门用于 Android 开发的工具包，里面有各种版本的开发框架和工具，以及丰富的文档，打开 http://developer.android.com/sdk/index.html 可以下载最新版本的针对 Windows 7 平台的 SDK。

当下载完成上述 3 个工具之后，需要对开发环境进行如下配置。

（1）配置 Eclipse

第 1 步：打开 Eclipse，在菜单栏中选择 help | Install New SoftWare 命令，弹出如图 1-23 所示的对话框。

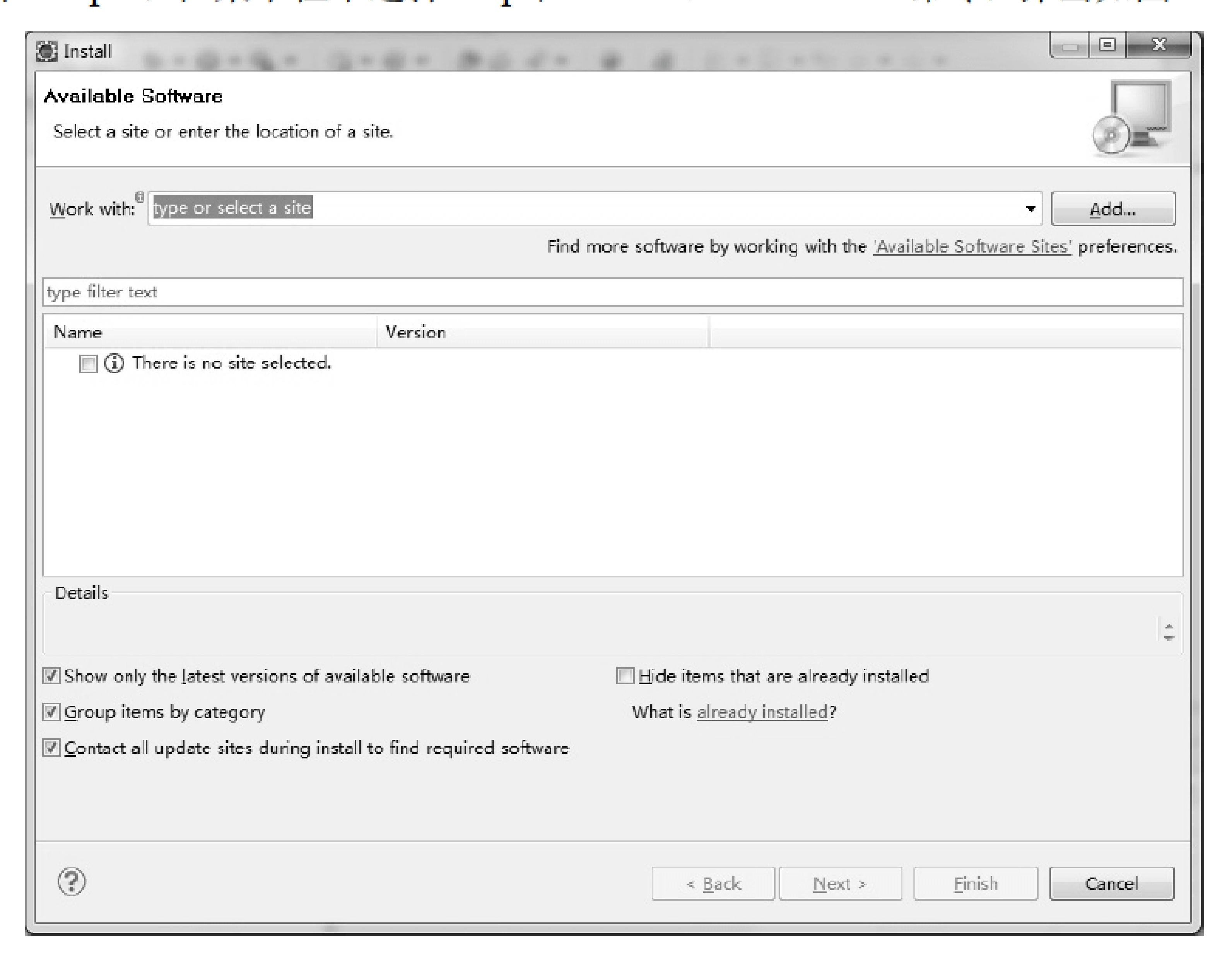
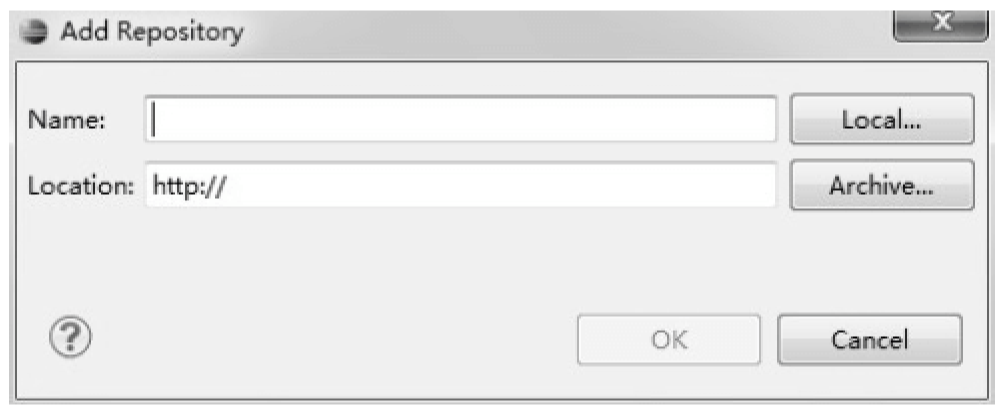


图 1-23　Install 对话框

第 2 步：单击 Add 按钮，弹出如图 1-24 所示的对话框。

第 3 步：在 Name 文本框中输入 Android 或者自定义的任何名字，在 Location 文本框中输入 https:// dl-ssl.google.com/android/eclipse/，输入后的效果如图 1-25 所示。



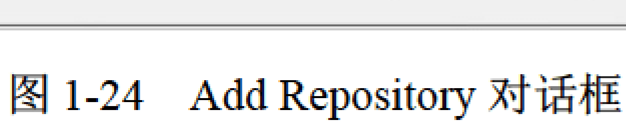


图 1-24　Add Repository 对话框

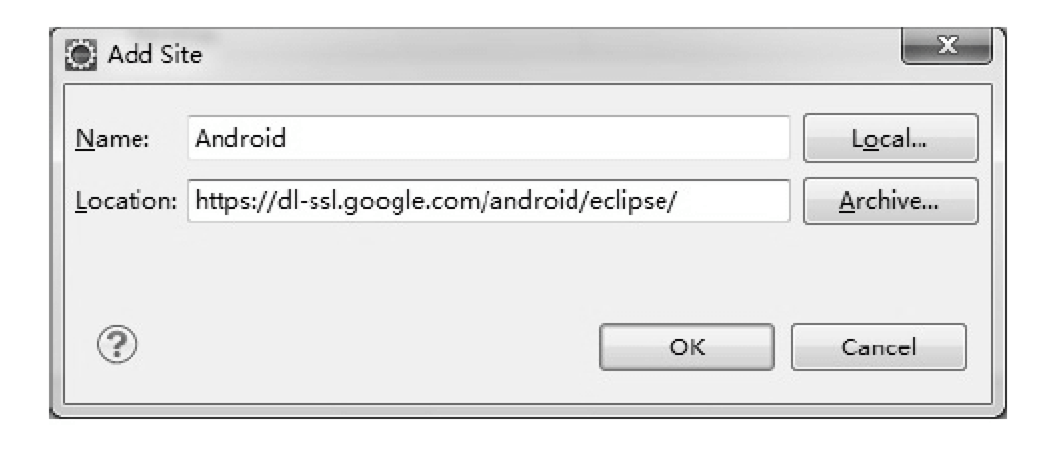
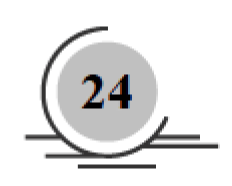


图 1-25　Add Site 对话框

第 4 步：如果发现 https://无法使用，可以改成 http://尝试一下，当输入好名字和地址之后，单击 OK 按钮，弹出如图 1-26 所示的对话框。

图 1-26 中的两个插件都是开发 Android 必不可少的工具包，Android DDMS 可以用来调试、管理 Android 进程、存储器、查看日志的工具；Android Development Tool 简称 ADT，是开发 Android 的插件，只有安装了 ADT 才能创建 Android 工程。



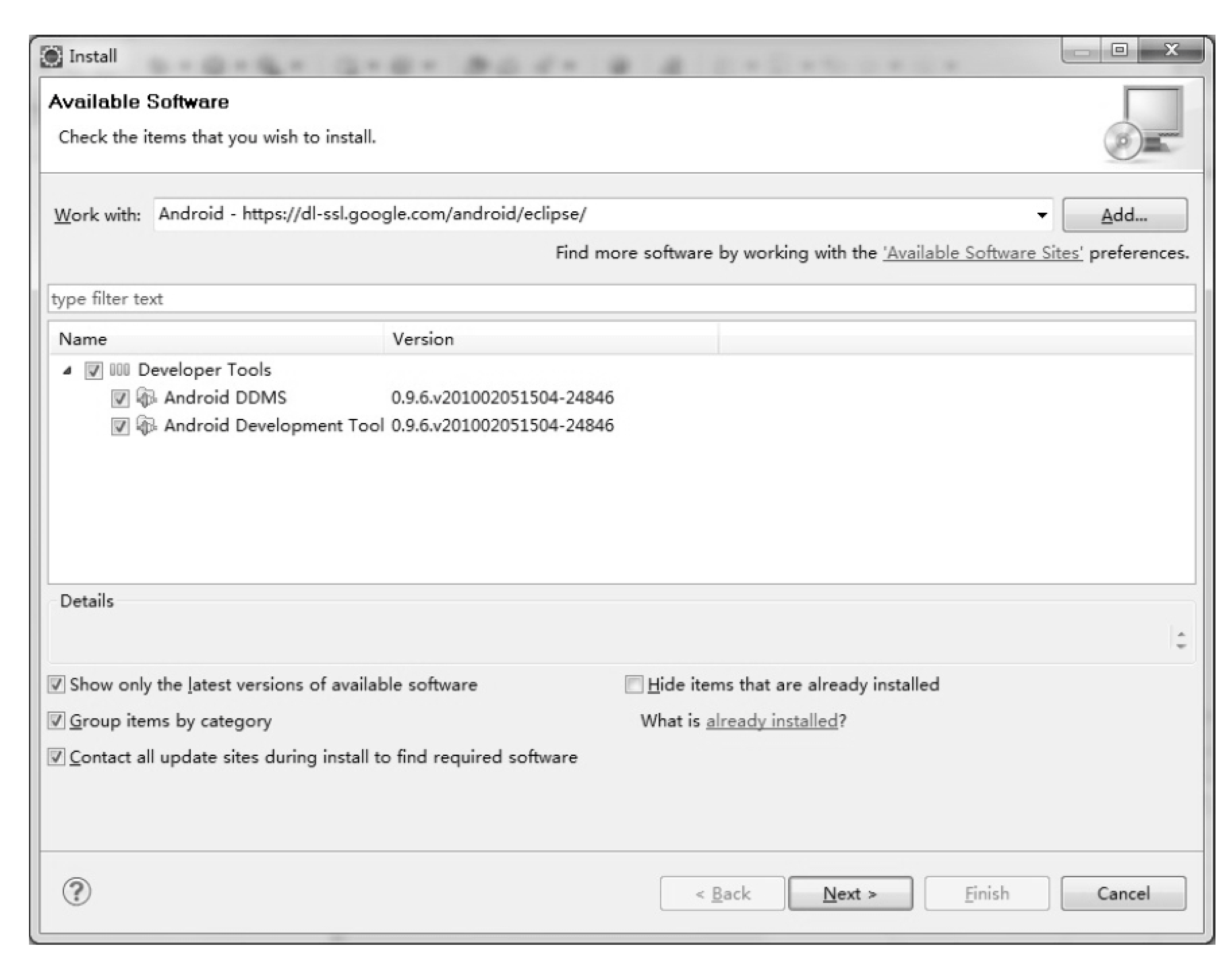


图 1-26　Available Software 界面

第 5 步：单击 Next 按钮，进入如图 1-27 所示的对话框。

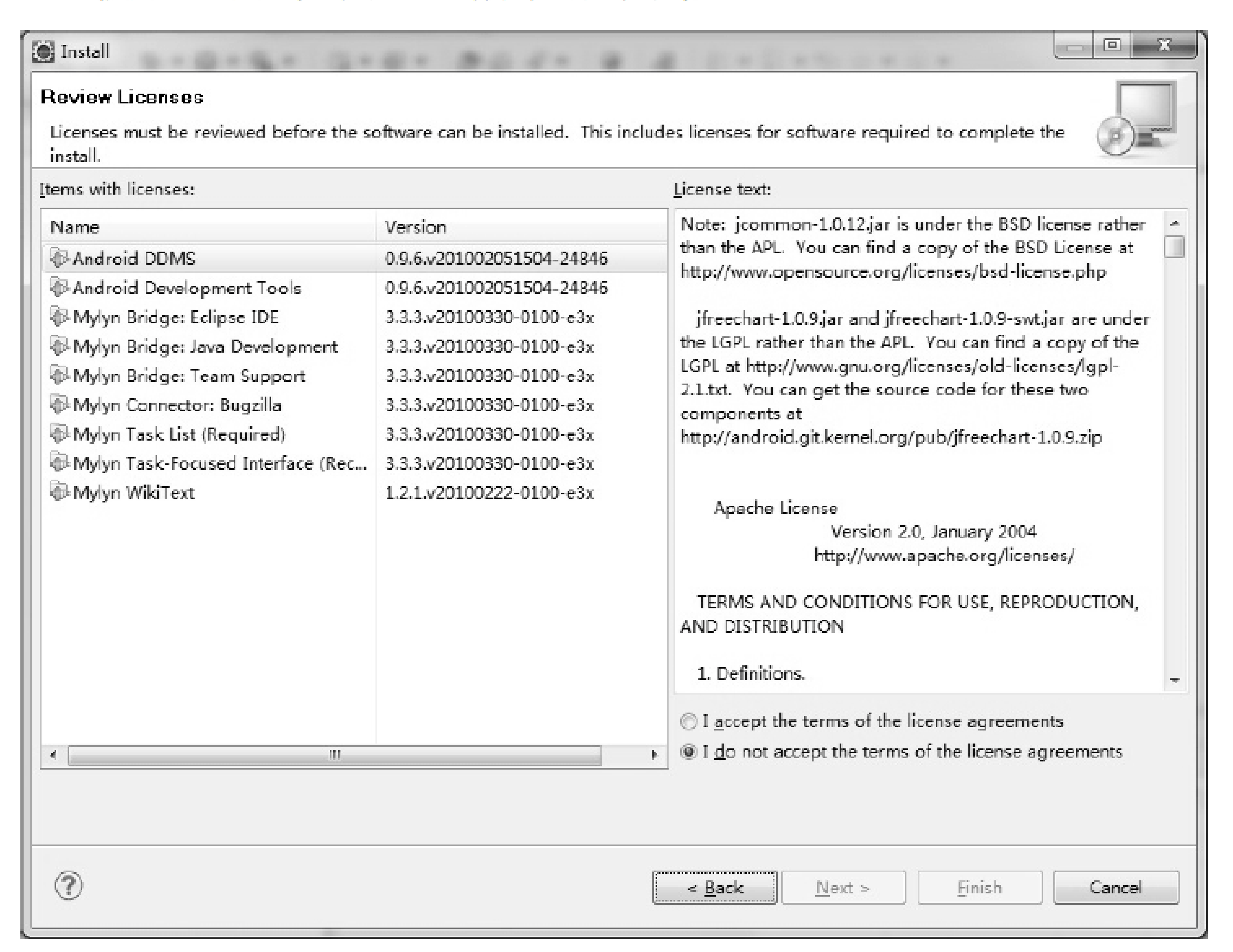


图 1-27　选择安装

图 1-27 中列出了将会安装的工具包，选中“I accept…”单选按钮，单击 Next 按钮开始安装插件，如图 1-28 所示。

第 6 步：当所有插件安装成功后，会弹出提示对话框，如图 1-29 所示。

这时需要单击 Yes 按钮重启 Eclipse 让所有插件生效。

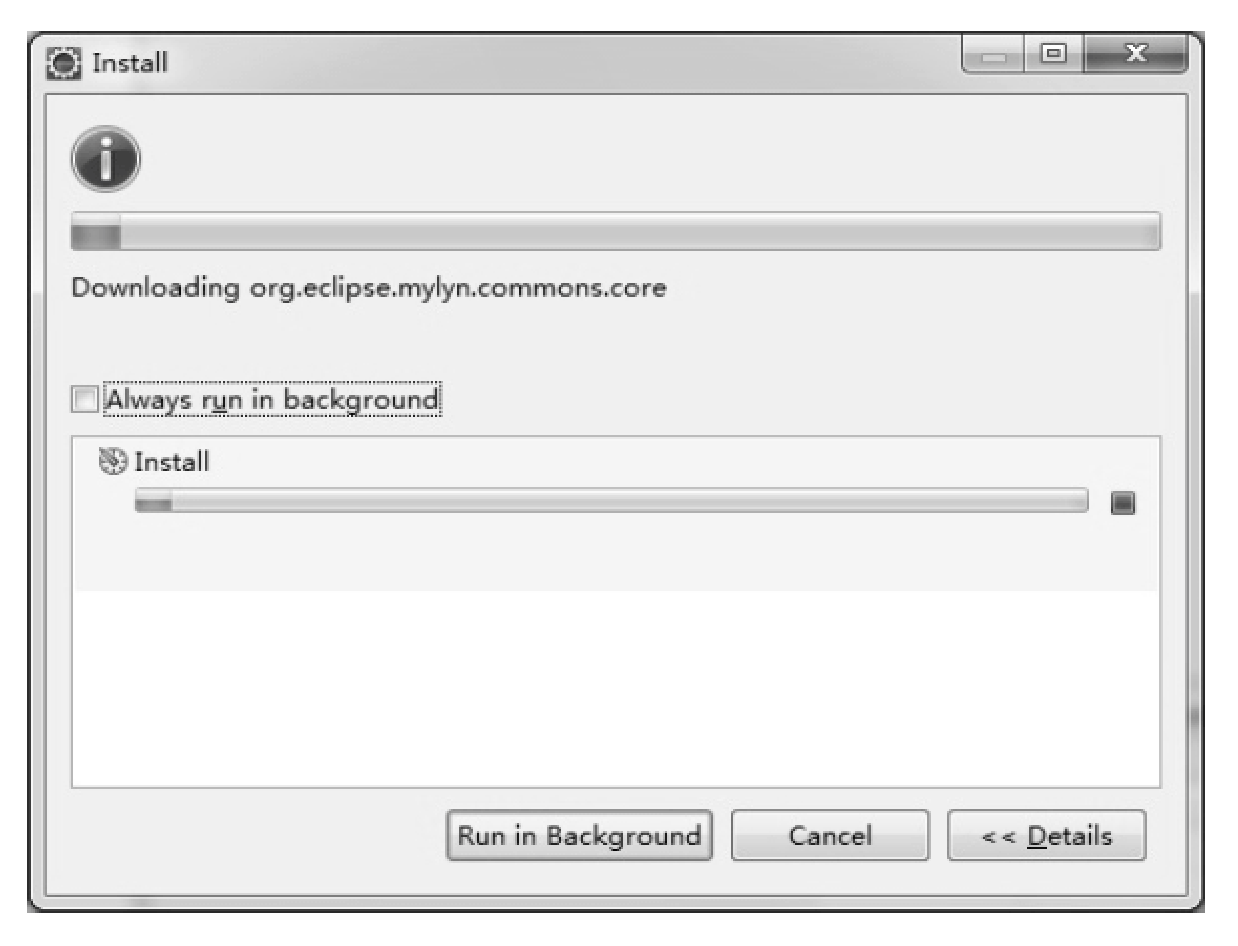


图 1-28　开始安装

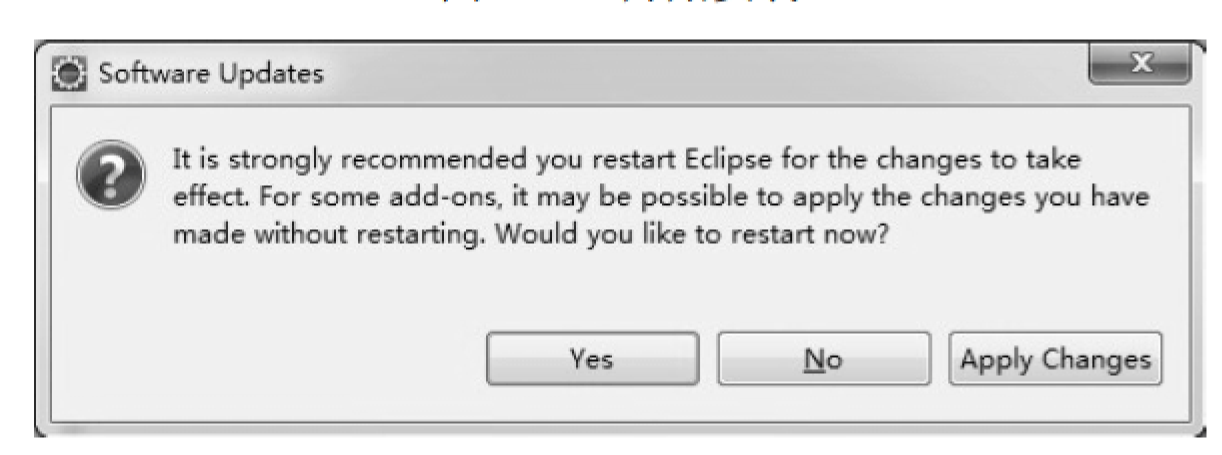


图 1-29　安装成功

（2）配置 Android SDK

打开 Eclipse，选择 Window | Preferences 命令，弹出如图 1-30 所示的对话框。

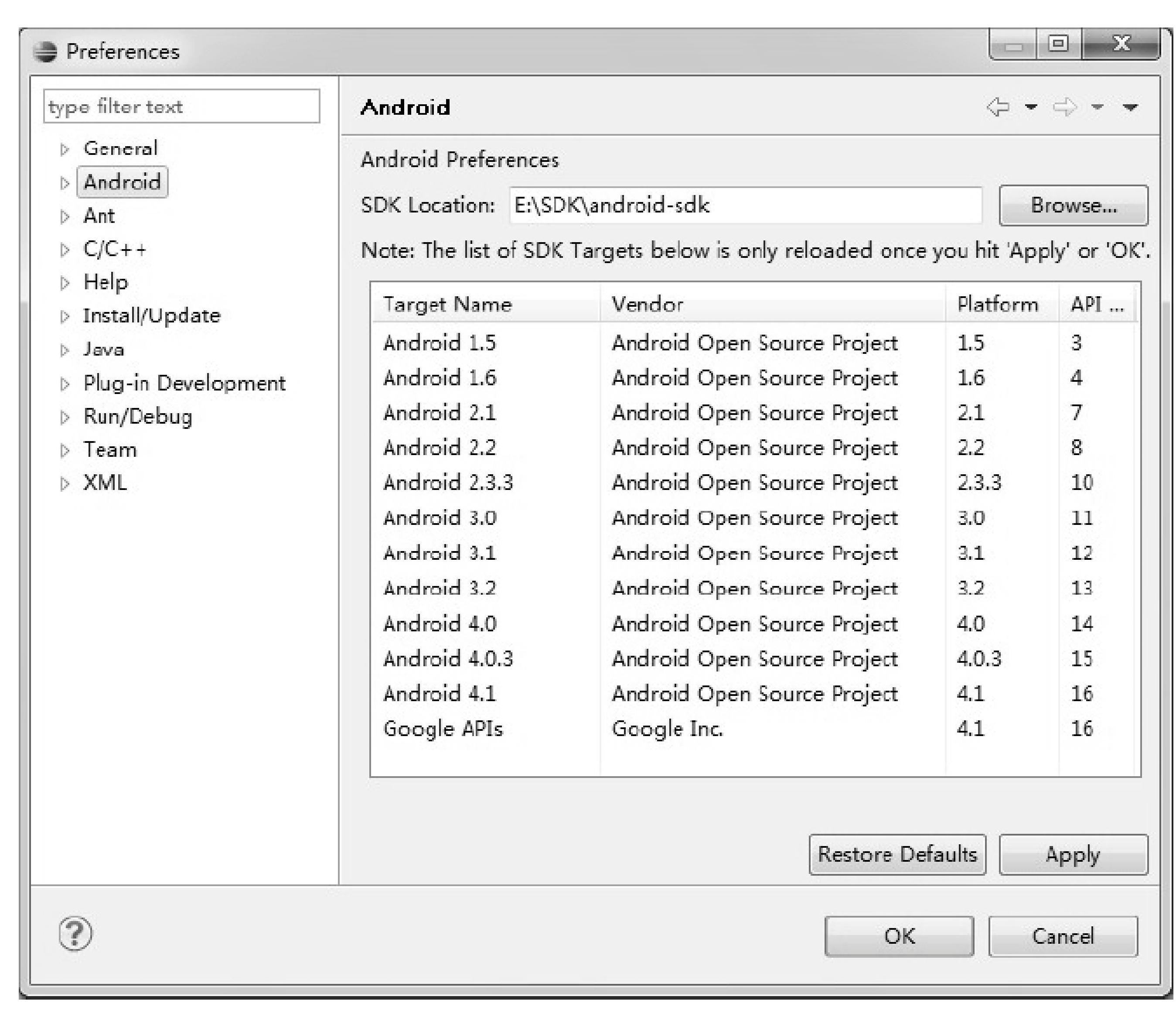


图 1-30　配置对话框

这样，就可以从 Eclipse 中新建 Android 工程，要想新建工程是基于什么版本的 Android 系统，可以打开 SDK 根目录下的 SDK 管理工具 SDK Manager.exe，双击后进入 SDK 工具包管理对话框，如图 1-31 所示。

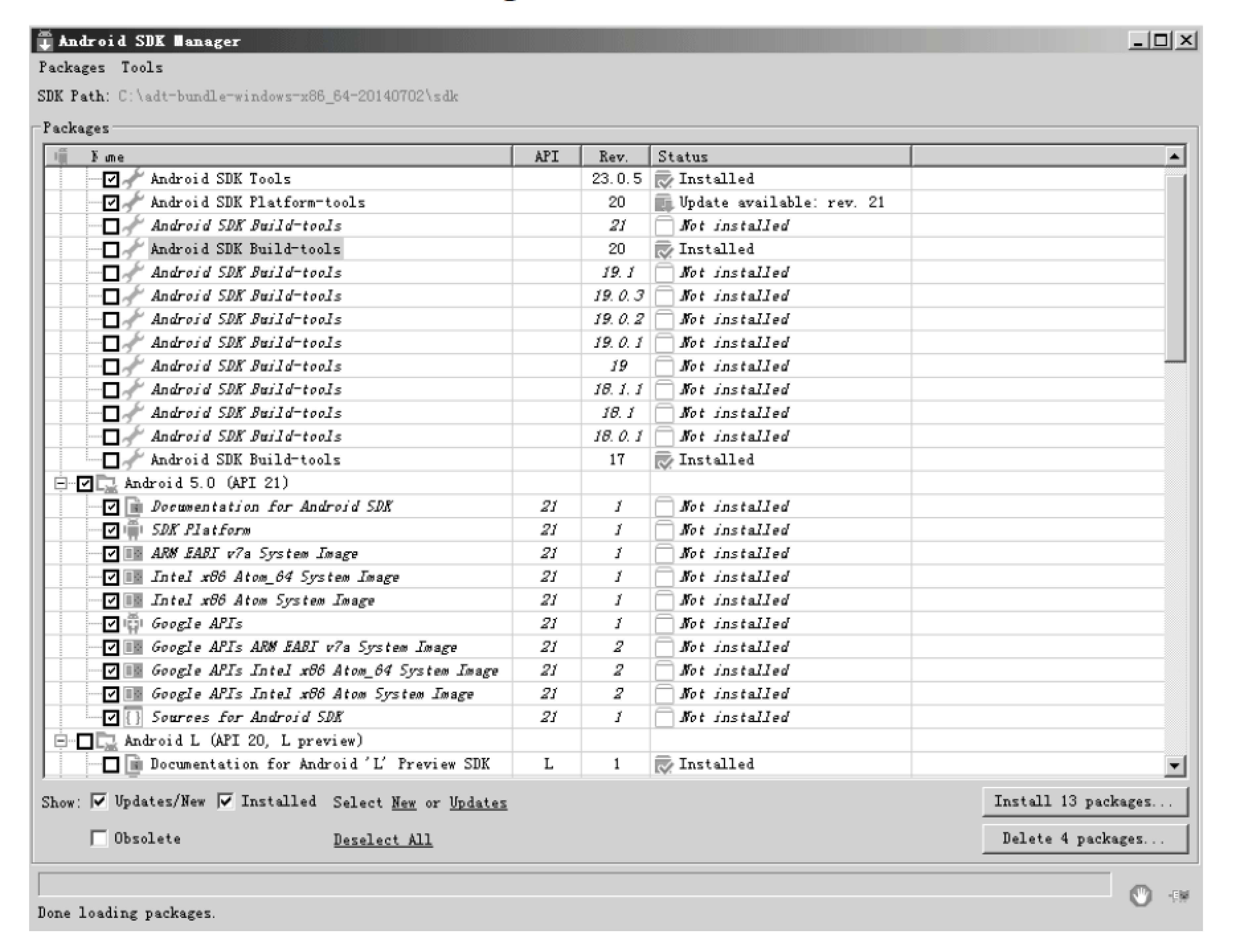


图 1-31　Android SDK 管理对话框

图 1-31 中很清晰地列出了当前版本 SDK 中包含的工具包，以及已经安装了的和没有安装的版本。可以继续单击 Install 13 Packages 或者 Delete 4 Packages 按钮安装和删除 SDK 中的工具包。如果是安装，则过程会比较慢，与网速的关系较大。当我们将 SDK 中的工具包安装完毕，同时也完成了 Eclipse 和 SDK 的配置工作，至此 Windows 7 平台下基于 SDK 的 Android 的开发环境搭建全部完成。

第2章　Android驱动开发基础

驱动含有推动和发动之意，计算机领域中的驱动也含有推动之意。例如在生活中总会遇到这样的场景：买了一个新USB鼠标，插在电脑上后会出现“安装新的驱动”的提示，买了一台新的打印机，也需要安装驱动后才能使用。本章将简要讲解Android驱动程序开发所必须具备的基本知识，为读者学习本书后面的知识打下基础。

2.1　驱动程序基础

在计算机世界中，驱动是一个应用程序。驱动程序是添加到操作系统中的一段代码，通常这段代码比较简短，但里面包含了和硬件相关的设备信息。有了这些信息，计算机就可以与设备进行通信，从而可以使用这些硬件。驱动程序是硬件厂商根据操作系统编写的配置文件，如果没有驱动程序，计算机中的硬件就无法正常工作，并且操作系统不同，对应的硬件驱动程序也不同。硬件厂商为了保证硬件的兼容性及增强硬件的功能，会不断更新、升级驱动程序，例如显卡芯片公司Nvidia平均每个月会升级驱动程序2～3次。本节将简要介绍驱动程序的基本知识及驱动开发所需要做的一些工作。

2.1.1　什么是驱动程序

驱动程序是硬件的一个构成部分，当我们安装新的硬件时必须安装对应的驱动程序。当安装一个原本不属于电脑或手机中默认的硬件设备时，系统会提示要求我们安装驱动程序，以将新的硬件与电脑或手机系统连接起来。驱动程序在此扮演了一个沟通的角色，负责把硬件的功能告诉电脑或手机系统，并且也将系统指令“开始工作”传达给硬件。

例如在Windows系统中，在安装主板、光驱、显卡、声卡这些硬件产品时，都会对应着一套完整的驱动程序，否则这些硬件将无法使用。在现实中最常见的就是打印机驱动程序，如果没有驱动程序，则电脑无法控制打印机完成打印工作。

在实际计算机应用中，一般可以通过如下3种途径得到驱动程序。

（1）购买的硬件附带有驱动程序。

（2）操作系统自带的驱动程序，例如Windows系统自带了大量的驱动程序。

（3）从Internet下载驱动程序，这种途径往往能够得到最新的驱动程序。

手机作为一种智能设备，也需要通过驱动程序来建立设备和外接硬件的连接，这种驱动通常被称为手机驱动。手机驱动是指有的手机和电脑不能直接连接，必须用手机自带的磁盘驱动一下。其实就是安装了一个读取手机内存信息的程序。也可以在网上找到安装，在网上搜索机型和驱动就能找到。而且在一部分手机中，通过数据线、蓝牙、红外方式连接电脑后还需要软件才能将数据传输到电脑，或者将数据传输到手机。此时可以使用所购手机的随机光盘中的驱动程序解决问题，也可以在手机网站或论坛上下载。

和Windows系统一样，在Android智能设备中，也经常需要使用一些外部硬件设备，例如蓝牙耳机、SD存储卡和摄像头等，要想使用这些外部辅助设备，也需要事先安装对应的驱动程序，本书的重点内容便是讲解在Android系统中开发驱动程序的基本知识，并剖析对驱动进行移植的知识。

2.1.2　驱动开发需要做的工作

Android 作为当前最流行的手机操作系统之一，受到了广大开发人员和商家的青睐，其正在蓬勃发展，带来了无限商机。其实 Android 是一个开放的系统，这个系统的体积非常庞大，开发人员无须掌握整个 Android 体系中的开发知识，只需熟悉其中某一个部分即可。从具体功能上来划分，Android 开发主要分为如下 3 个领域。

（1）系统移植

移植开发的目的是构建硬件系统，并且移植 Android 的软件系统，最终形成手机产品。

（2）Android 应用程序开发

Android 的应用程序开发是 Android 开发的另一个方面，从开发的角度来看，这种形式的开发可以基于某个硬件系统。在没有硬件系统的情况下，也可以基于 Linux 或者 Windows 下的 Android 模拟器进行开发，这种类型的开发工作发生在 Android 系统的上层。应用程序开发的目的是开发出各种 Android 应用程序，然后将这些应用程序投入 Android 市场进行交易。

在 Android 软件系统中，Java 框架和 Java 应用之间的接口也就是 Android 的系统接口（系统 API）。这个层次是标准的接口，所有的 Android 应用程序都是基于这个层次的接口开发出来的。Android 系统中的 Java 应用是一组内置的 Android 应用程序。

作为 Android 应用程序开发者，其开发的应用程序其实和 Android 系统的 Java 应用层次的应用程序是一个层次的内容。例如，Android 系统提供了基本的桌面程序，开发者可以根据 Android 的系统接口，实现另外一个桌面程序，提供给用户安装使用；根据 Android 系统的接口开发游戏，也是 Android 应用程序开发的一个重要方向。

上述两种类型的开发结构如图 2-1 所示。

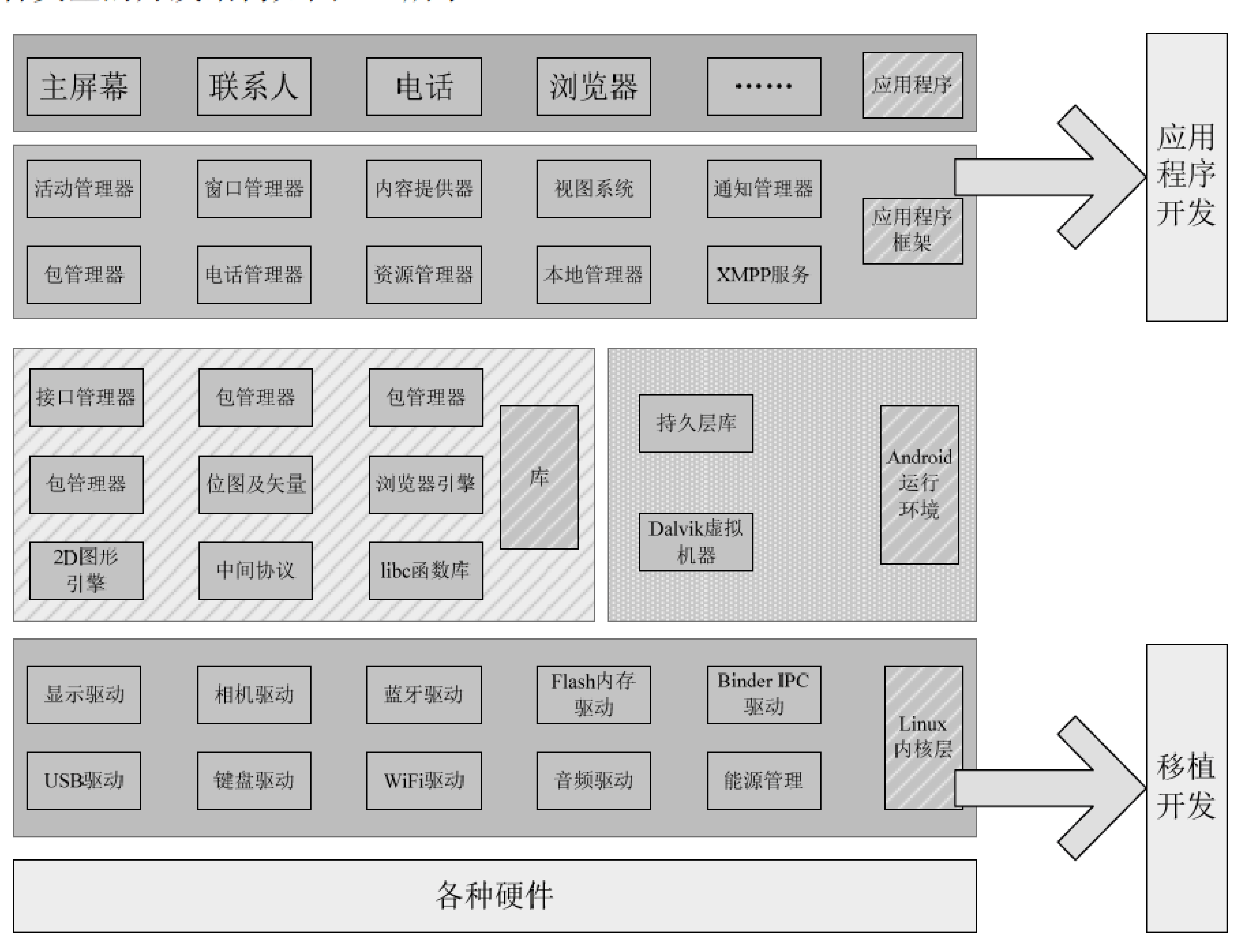


图 2-1　Android 开发的领域

（3）Android 系统开发

系统开发的目的是升级或改造 Android 中已经存在的应用和架构，开发出自己特色的手机系统。例如联想手机乐 Phone 就是在 Android 基础上打造的一款适合国人习惯的手机系统，如图 2-2 所示。



图 2-2　乐 Phone

Android 系统开发的一个比较典型的示例就是当系统需要某种功能时，为了给 Java 层次的应用程序提供调用的接口，需要从底层到上层的整体开发，具体步骤如下。

- ☑ 增加 C 或者 C++本地库。
- ☑ 定义 Java 层所需要的类（系统 API）。
- ☑ 将所需要的代码封装成 JNI。
- ☑ 结合 Java 类和 JNI。
- ☑ 应用程序调用 Java 类。

一定要慎重对待对 Android 系统 API 的改动工作，因为系统 API 的稍微变动就可能会涉及 Android 应用程序的兼容问题。

Android 系统本身的功能也在增加和完善的过程中，因此 Android 系统的开发也是一个重要的方面，这种类型的开发会涉及 Android 软件系统的各个层次。在更多的时候，Android 系统开发只是在不改变系统 API 的情况下修正系统的缺陷，增加系统的稳定性。

从商业模式的角度来看，第一种类型的开发和第二种类型的开发是 Android 开发的主流。事实上，移动电话的制造者主要进行第一种类型的开发，产品是 Android 实体手机。公司、个人和团体都可以进行第二种类型的开发，其产品是不同的 Android 应用程序。

在 Android 的开发过程中，每一种类型的开发都只涉及整个 Android 系统的一个子集。在 Android 系统中，有着众多的开发点，这些开发点相互独立，又有内在联系。在开发的过程中，只需重点掌握目前开发点涉及的内容即可。

2.2　Linux 开发基础

Linux 是一类 UNIX 计算机操作系统的统称。Linux 操作系统的内核的名字也是 Linux。Linux 操作系统也是自由软件和开放源代码发展中最著名的例子。严格来讲，Linux 这个词本身只表示 Linux 内核，但在实际上人们已经习惯了用 Linux 来形容整个基于 Linux 内核，并且使用 GNU 工程各种工具和数据库的操作系统。本节将简要介绍 Linux 系统的基本知识。

2.2.1　Linux 简介

Linux 最早开始于一位名叫 Linus Torvalds 的计算机业余爱好者，当时他是芬兰赫尔辛基大学的学生。他的目的是想设计一个代替 Minix（是由一位名叫 Andrew Tannebaum 的计算机教授编写的一个操作系统示教程序）的操作系统，这个操作系统可用于 386、486 或奔腾处理器的个人计算机上，并且具有 UNIX 操作系统的全部功能，因而开始了 Linux 雏形的设计。

1983 年，理查德·马修·斯托曼（Richard Stallman）创立了 GNU 计划（GNU Project），目标是为了发展一个完全免费自由的 Unix-like 操作系统。自 1990 年发起这个计划以来，GNU 开始大量地生产或收集各种系统所必备的元件，例如函式库（libraries）、编译器（compilers）、侦错工具（debuggers）、文字编

辑器（text editors）、网页服务器（web server），以及一个 UNIX 的使用者接口（UNIX shell）——除了执行核心（kernel）仍然欠缺。1990 年，GNU 计划开始在马赫微核（Mach microkernel）的架构之上开发系统核心，也就是所谓的 GNU Hurd，但是这个基于 Mach 的设计异常复杂，发展进度相对缓慢。

2.2.2　Linux 的发展趋势

随着 SaaS、云计算、虚拟化、移动平台、企业 2.0 等新兴技术的发展，Linux 事业正面临巨大的发展机遇，主要体现在如下 3 个方面。

（1）向企业级核心应用迈进

Linux 的采用已由网络领域逐步转向了关键业务应用，企业关键任务成为 IBM 的增长领域，例如 ERP 软件。同时，随着 IT 决策逐步从 IT 主管下达给 IT 管理人员，这些管理者对 Linux 显示了强烈的支持，但同时对安全、可用性与服务提出了更高的需求。尽管很多用户仍将 UNIX 视为关键任务的平台，但随着 Linux 开发者逐步缩小两者的功能性差距，有越来越多的用户开始将关键业务部署在 Linux 之上。

（2）Linux 将主导移动平台

Linux 进入到移动终端操作系统后，很快就以其开放源代码的优势吸引了越来越多的终端厂商和运营商对它的关注，包括摩托罗拉和 NTT DoCoMo 等知名的厂商。已经开发出的基于 Linux 的手机有摩托罗拉的 zn5、V8、A1210、A810、A760、A768、A780、e680i、e680、e2、e680g、E6、E8、em30、CEC 的 e2800、三星的 i519 等。

Linux 与其他操作系统相比算是后来者，但 Linux 具有两个其他操作系统无法比拟的优势。其一，Linux 具有开放的源代码，能够大大降低成本。其二，既满足了手机制造商根据实际情况有针对性地开发自己的 Linux 手机操作系统的要求，又吸引了众多软件开发商对内容应用软件的开发，丰富了第三方应用。

（3）新技术为 Linux 加速

在目前的企业级计算领域，云计算、SaaS、虚拟化是热门技术话题。在这些领域，Linux 同样大有可为。云计算将全部使用 Linux，Linux 也是一款未来运行数据中心虚拟机的理想操作系统。

2.2.3　Android 基于 Linux 系统

在了解 Linux 和 Android 两者之间的关系之前，首先需要明确如下 3 点。

（1）Android 采用 Linux 作为内核。

（2）Android 对 Linux 内核做了修改，以适应其在移动设备上的应用。

（3）Andorid 开始是作为 Linux 的一个分支，后来由于无法并入 Linux 的主开发树，曾经被 Linux 内核组从开发树中删除。2012 年 5 月 18 日，Linux kernel 3.3 版本发布后又被加入。

Android 是在 Linux 的内核基础之上运行的，提供的核心系统服务包括安全、内存管理、进程管理、网络组和驱动模型等内容。内核部分还相当于一个介于硬件层和系统中其他软件组之间的一个抽象层次。但是严格来说它不算是 Linux 操作系统。

因为 Android 内核是由标准的 Linux 内核修改而来的，所以继承了 Linux 内核的诸多优点，保留了 Linux 内核的主题架构。同时 Android 按照移动设备的需求，在文件系统、内存管理、进程间通信机制和电源管理方面进行了修改，添加了相关的驱动程序和必要的新功能。但是和其他精简的 Linux 系统（如 uClinux）相比，Android 很大程度地保留了 Linux 的基本架构，因此 Android 的应用性和扩展性更强。当前 Android 版本对应的 Linux 内核版本如下。

☑　Android 1.5：Linux-2.6.27。

☑ Android 1.6：Linux-2.6.29。
☑ Android 2.0,2.1：Linux-2.6.29。
☑ Android 2.2：Linux-2.6.32.9。
☑ ……
☑ Android 4.3：Linux-3.4。
☑ Android 4.4：Linux-3.12。

2.2.4　Android 和 Linux 内核的区别

Android 系统的系统层面的底层是 Linux，中间加上了一个叫做 Dalvik 的 Java 虚拟机和 ART 运行环境，表面层上面是 Android 运行库。每个 Android 应用都运行在自己的进程上，享有 ART 或 Dalvik 虚拟机为它分配的专有实例。为了支持多个虚拟机在同一个设备上高效运行，Dalvik 被改写过。

Dalvik 虚拟机执行的是 Dalvik 格式的可执行文件（.dex）——该格式经过优化，以降低内存耗用到最低。Java 编译器将 Java 源文件转化为 class 文件，class 文件又被内置的 dx 工具转化为 dex 格式文件，这种文件在 Dalvik 虚拟机上注册并运行。

Android 系统的应用软件都是运行在 Dalvik 之上的 Java 软件，而 Dalvik 是运行在 Linux 中的，在一些底层功能——例如线程和低内存管理方面，Dalvik 虚拟机是依赖 Linux 内核的。由此可见，可以说 Android 是运行在 Linux 之上的操作系统，但是它本身不能算是 Linux 的某个版本。

Android 内核和 Linux 内核的差别主要体现在 11 个方面，下面将一一进行简要介绍。

（1）Android Binder

Android Binder 的源代码位于 drivers/staging/android/binder.c。

Android Binder 是基于 OpenBinder 框架的一个驱动，用于提供 Android 平台的进程间通信（Inter Process Communication，IPC）。原来的 Linux 系统上层应用的进程间通信主要是 D-bus（Desktop bus），采用消息总线的方式来进行 IPC。

（2）Android 电源管理（PM）

Android 电源管理是一个基于标准 Linux 电源管理系统的轻量级的 Android 电源管理驱动，针对嵌入式设备做了很多优化。利用锁和定时器来切换系统状态，控制设备在不同状态下的功耗，以达到节能的目的。

Android 电源管理的源代码分别位于如下位置。

☑ kernel/power/earlysuspend.c
☑ kernel/power/consoleearlysuspend.c
☑ kernel/power/fbearlysuspend.c
☑ kernel/power/wakelock.c
☑ kernel/power/userwakelock.c

Android 5.0 版本将引用 JobScheduler 调度程序，好处是增加设备续航时间，以达到节省电量的目的。

（3）低内存管理器（Low Memory Killer）

Android 中的低内存管理器和 Linux 标准的 OOM（Out Of Memory）相比，其机制更加灵活，它可以根据需要杀死进程来释放需要的内存。Low Memory Killer 的代码很简单，关键的一个函数是 Lowmem_shrinker。作为一个模块在初始化时调用 register_shrinke 注册了一个 lowmem_shrinker，它会被 vm 在内存紧张的情况下调用。Lowmem_shrinker 用来完成具体操作。简单地说就是寻找一个最合适的进程杀死，从而释放它占用的内存。

低内存管理器的源代码位于 drivers/staging/android/lowmemorykiller.c。

（4）匿名共享内存（Ashmem）

匿名共享内存为进程间提供大块共享内存，同时为内核提供回收和管理这个内存的机制。如果一个程序尝试访问 Kernel 释放的一个共享内存块，它将会收到一个错误提示，然后重新分配内存并重载数据。

匿名共享内存的源代码位于 mm/ashmem.c。

（5）Android PMEM（Physical）

PMEM 用于向用户空间提供连续的物理内存区域，DSP 和某些设备只能工作在连续的物理内存上。驱动中提供了 mmap、open、release 和 ioctl 等接口。

Android PMEM 的源代码位于 drivers/misc/pmem.c。

（6）Android Logger

Android Logger 是一个轻量级的日志设备，用于抓取 Android 系统的各种日志，是 Linux 所没有的。

Android Logger 的源代码位于 drivers/staging/android/logger.c。

（7）Android Alarm

Android Alarm 提供了一个定时器用于把设备从睡眠状态唤醒，同时它也提供了一个即使在设备睡眠时也会运行的时钟基准。

Android Alarm 的源代码位于 drivers/rtc/alarm.c 和 drivers/rtc/alarm-dev.c。

（8）USB Gadget 驱动

USB Gadget 驱动是一个基于标准 Linux USB gadget 驱动框架的设备驱动，Android 的 USB 驱动是基于 gadget 框架的。

USB Gadget 驱动的源代码位于如下位置。

☑ drivers/usb/gadget/android.c
☑ drivers/usb/gadget/f_adb.c
☑ drivers/usb/gadget/f_mass_storage.c

（9）Android Ram Console

为了提供调试功能，Android 允许将调试日志信息写入一个被称为 RAM Console 的设备中，它是一个基于 RAM 的 Buffer。

Android Ram Console 的源代码位于 drivers/staging/android/ram_console.c。

（10）Android timed device

Android timed device 提供了对设备进行定时的控制功能，目前仅支持 vibrator 和 LED 设备。

Android timed device 的源代码位于 drivers/staging/android/timed_output.c（timed_gpio.c）。

（11）Yaffs2 文件系统

在 Android 系统中，采用 Yaffs2 作为 MTD NAND Flash 文件系统。Yaffs2 是一个快速稳定的应用于 NAND 和 NOR Flash 的跨平台的嵌入式设备文件系统，同其他 Flash 文件系统相比，Yaffs2 使用更小的内存来保存它的运行状态，因此占用内存小；Yaffs2 的垃圾回收非常简单而且快速，因此能达到更好的性能；Yaffs2 在大容量的 NAND Flash 上性能表现尤为明显，非常适合大容量的 Flash 存储。

Yaffs2 文件系统源代码位于 fs/yaffs2/。

Android 是在 Linux 的内核基础之上运行的，提供的核心系统服务包括安全、内存管理、进程管理、网络组和驱动模型等内容。内核部分还相当于一个介于硬件层和系统中其他软件组之间的一个抽象层次。但是严格来说它不算是 Linux 操作系统。

Android 中的 Linux 内核与驱动结构如图 2-3 所示。

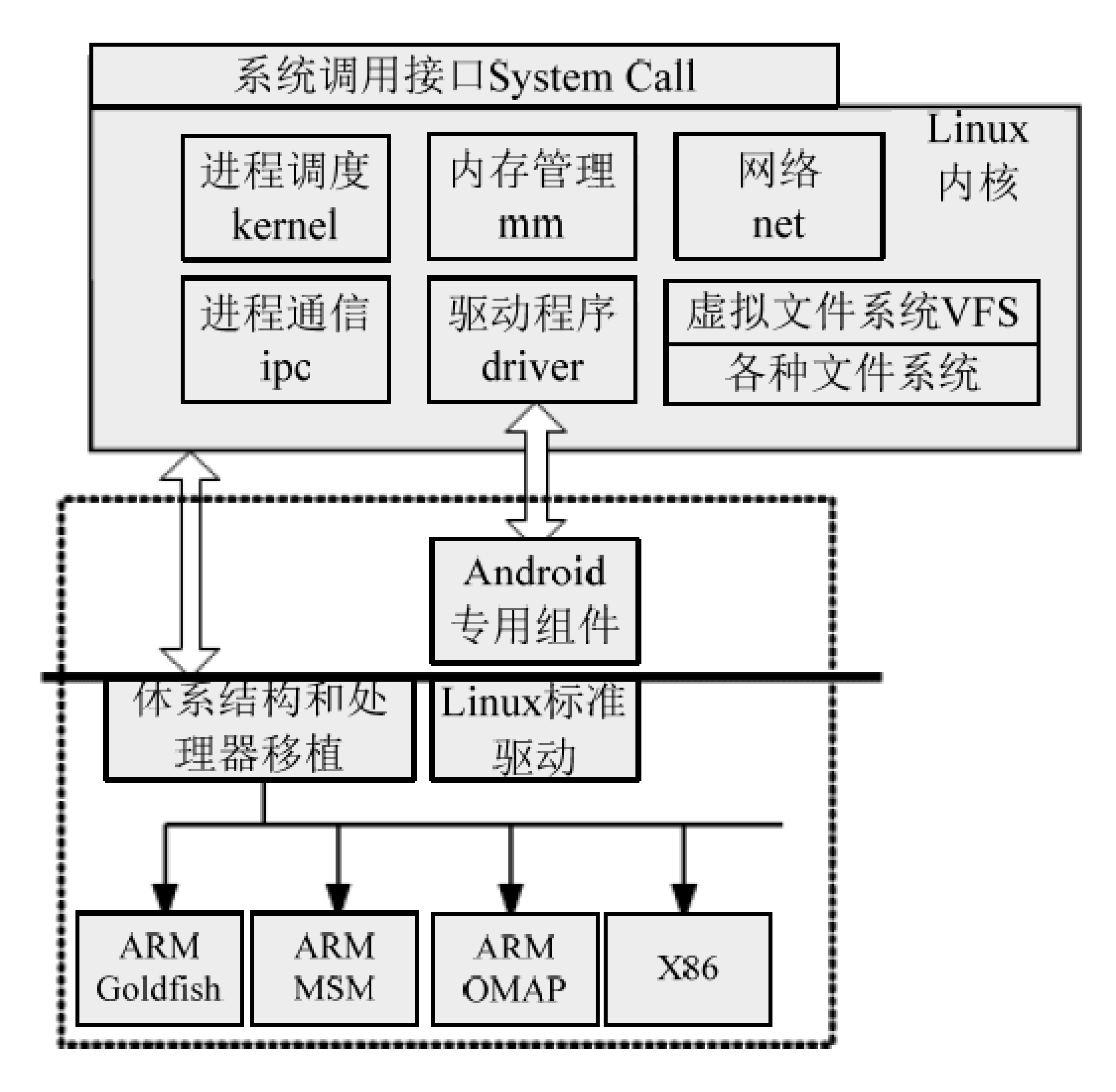


图 2-3　Android 中的 Linux 内核与驱动结构

2.2.5　Android 独有的驱动

经过本书前面内容的学习可知，Android 驱动是离不开 Linux 驱动的，但是 Android 并没有完全照搬 Linux 系统内核，除了对 Linux 进行部分修正外还增加了不少内容。Android 驱动主要分两种类型：Android 专有驱动和 Android 使用的设备驱动（Linux）。

（1）Android 专有驱动程序

Android Ashmem 匿名共享内存；为用户空间程序提供分配内存的机制，为进程间提供大块共享内存，同时为内核提供回收和管理这个内存。

☑　Android Logger：轻量级的 LOG（日志）驱动。
☑　Android Binder：基于 OpenBinder 框架的一个驱动。
☑　Android Power Management：电源管理模块。
☑　Low Memory Killer：低内存管理器。
☑　Android PMEM：物理内存驱动。
☑　USB Gadget：USB 驱动（基于 gaeget 框架）。
☑　Ram Console：用于调试写入日志信息的设备。
☑　Time Device：定时控制设备。
☑　Android Alarm：硬件时钟。

（2）Android 上的设备驱动

☑　Framebuff 显示驱动。
☑　Event：输入设备驱动。
☑　ALSA：音频驱动。
☑　OSS：音频驱动。
☑　v412 摄像头：视频驱动。
☑　MTD 驱动。
☑　蓝牙驱动。

☑ WLAN 设备驱动。

2.2.6　为 Android 构建 Linux 的操作系统

如果我们以一个原始的 Linux 操作系统为基础，改造成为一个适合于 Android 的系统，所做的工作其实非常简单，就是增加适用于 Android 的驱动程序。在 Android 中有很多 Linux 系统的驱动程序，将这些驱动程序移植到新系统的步骤非常简单，具体来说有以下 3 个步骤。

（1）编写新的源代码。

（2）在 KConfig 配置文件中增加新内容。

（3）在 Makefile 中增加新内容。

在 Android 系统中，通常会使用 FrameBuffer 驱动、Event 驱动、Flash MTD 驱动、WiFi 驱动、蓝牙驱动和串口等驱动程序，并且还需要音频、视频、传感器等驱动和 sysfs 接口。移植的过程就是移植上述驱动的过程，我们的工作是在 Linux 下开发适用于 Android 的驱动程序，并移植到 Android 系统。

在 Android 中添加扩展驱动程序的基本步骤如下。

（1）在 Linux 内核中移植硬件驱动程序，实现系统调用接口。

（2）把硬件驱动程序的调用在 HAL 中封装成 Stub。

（3）为上层应用的服务实现本地库，由 Dalv 默认虚拟机调用本地库来完成上层 Java 代码的实现。

（4）最后编写 Android 应用程序，提供 Android 应用服务和用户操作界面。

2.3　Linux 内核结构

既然 Android 内核与 Linux 内核有密切的联系，所以在学习 Android 底层驱动开发之前必须了解 Linux 内核的基本知识。本节将详细讲解 Linux 内核架构的基本知识。

2.3.1　Linux 内核的体系结构

如图 2-4 所示为一个完整操作系统最基本的视图，由此可见，内核的作用就是将应用程序和硬件分离开。

内核的主要任务是负责与计算机硬件进行交互，实现对硬件的编程控制和接口操作，调度对硬件资源的访问。除此之外，内核为用户应用程序提供一个高级的执行环境和访问硬件的虚拟接口。其实提供硬件的兼容性是内核的设计目标之一，几乎所有的硬件都可以得到 Linux 的支持，只要不是为其他操作系统所定制。

与硬件兼容性相关的是可移植性，即在不同的硬件平台上运行 Linux 的能力。从最初只支持标准 IBM 兼容机上的 Intel X86 架构到现在可以支持 Alpha、ARM、MIPS、PowerPC 等几乎所有硬件平台，如此广泛的平台支持之所以能够成功，部分原因在于内核清晰地划分为了体系相关部分和体系无关部分。

再看图 2-5，这是 Linux 操作系统的基本视图。由此可见，Linux 内核分为如下两部分。

（1）体系相关部分：这部分内核为体系结构和硬件所特有。

（2）体系无关部分：这部分内核是可移植的。体系无关部分通常会定义与体系相关部分的接口，这样，内核向新的体系结构移植的过程就变成确认这些接口的特性并将它们加以实现的过程。

用户应用程序和内核之间的联系是通过它和内核的中间层——标准 C 库来实现，标准 C 库函数是建立在内核提供的系统调用基础之上的。通过标准 C 库，以及内核体系无关部分与体系相关部分的接口，用户

应用程序和部分内核都成为可移植的。根据上述描述，下面给出 Linux 操作系统的标准视图，具体如图 2-6 所示。

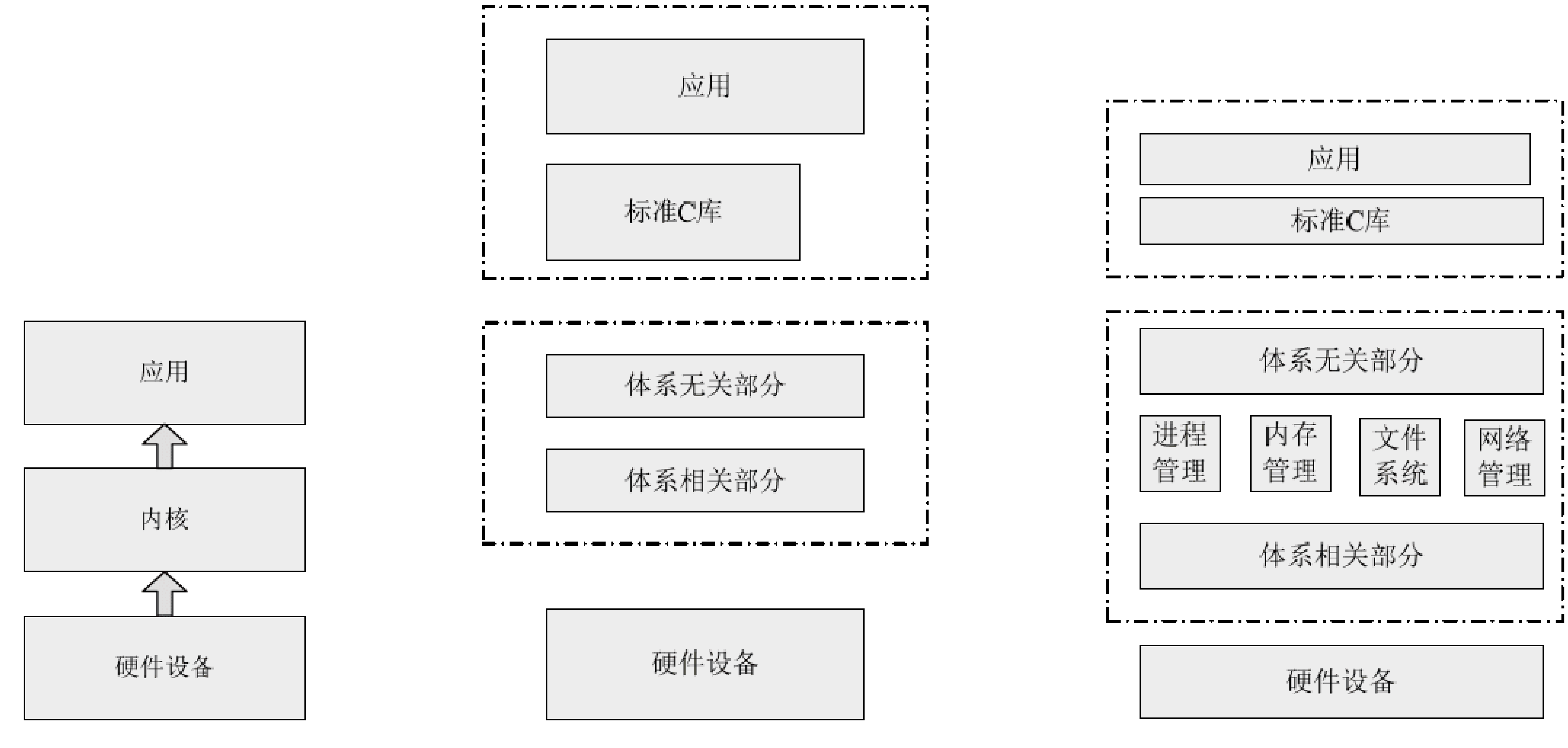


图 2-4　操作系统的基本视图　　图 2-5　Linux 操作系统的基本视图　　图 2-6　Linux 系统的标准视图

在上述 Linux 系统的标准视图中，主要构成模块的具体说明如下。

（1）系统调用接口

为了与用户应用程序进行交互，内核提供了一组系统调用接口，应用程序通过这组接口可以访问系统硬件和各种操作系统资源。系统调用接口层在用户应用程序和内核之间添加了一个中间层，在此扮演了一个函数调用多路复用和多路分解器的角色。

（2）进程管理

进程管理负责创建和销毁进程，并处理它们之间的互相联系（进程间通信），同时负责安排调度它们去分享 CPU。进程管理部分实现了一个进程世界的抽象，这个进程世界类似于我们的人类世界，只不过我们人类世界里的个体是人，而在进程世界里则是一个一个的进程，人与人之间通过书信、手机、网络等进行交互，而各个进程之间则是通过不同方式的进程间通信，我们所有人都在共享同一个地球，而所有进程都在共享一个或多个 CPU。

（3）内存管理

在进程世界里，内存是重要的资源之一。因此，管理内存的策略与方式是决定系统性能的一个关键因素。内核的内存管理部分根据不同的需要，提供了包括 malloc/free 在内的许多简单或者复杂的接口，并为每个进程都提供了一个虚拟的地址空间，基本上实现虚拟内存对进程的按需分配。

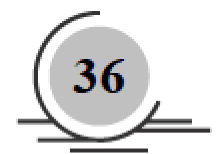
（4）虚拟文件系统

虚拟文件系统为用户空间提供了文件系统接口，同时又为各个具体的文件系统提供了通用的接口抽象。在 VFS 上面，是对诸如 open、close、read 和 write 之类函数的一个通用 API 抽象，在 VFS 下面则是具体的文件系统，它们定义了上层函数的实现方式。

通过虚拟文件系统，我们可以利用标准的 Linux 文件系统调用对不同介质上的不同文件系统进行操作。应该说，VFS 是内核在各种具体的文件系统上建立的一个抽象层，它提供了一个通用的文件系统模型，而该模型囊括了我们所能想到的所有文件系统的行为。

（5）网络功能

网络子系统处理数据包的收集、标识、分发、路由和地址的解析等所有网络有关的操作。socket 层是网



络子系统的标准 API，它为各种网络协议提供了一个用户接口。

（6）设备驱动程序

操作系统的目的是为用户提供一种方便访问硬件的途径，因此，几乎每一个系统操作最终都会映射到物理的硬件设备上。除了 CPU、内存等有限的几个对象，所有设备的访问控制操作都要由相关的代码来完成，这些代码就是所谓的设备驱动程序。

（7）代码

这里的代码需要依赖体系结构，因为部分内核代码是体系相关的，在“./linux/arch”子目录中定义了内核源代码中依赖于体系结构的部分，其中包含了对应各种特定体系结构的子目录。例如，对于一个典型的桌面系统来说，使用的是 i386 目录。

每个特定体系结构对应的子目录又包含了很多下级子目录，分别关注内核中的一个特定方面，例如引导、内核、内存管理等。

2.3.2　和 Android 驱动开发相关的内核知识

Android 是在 Linux 内核基础之上运行的，提供的核心系统服务包括安全、内存管理、进程管理、网络组和驱动模型等内容，下面简要讲解上述核心系统服务的基本知识。

1．安全

关于 Linux 安全的知识主要涉及了用户权限问题和目录权限问题。

（1）Linux 系统中用户和权限

Linux 系统中的每个文件和目录都有访问权限，用它来确定谁可以通过何种方式对文件和目录进行访问和操作。Linux 系统中规定了文件 3 种不同类型的用户：文件拥有者用户（user）、同组用户（group）、可以访问系统的其他用户（others）。并规定 3 种访问文件或目录的方式：读（r）、写（w）、可执行或查找（x）。

（2）文件及目录权限的功能

- ☑ 读权限（r）：表示只允许指定用户读取相应文件的内容，禁止对它做任何的更改操作，例如目录读权限表示可以列出存储在该目录下的文件，即读目录内容。
- ☑ 写权限（w）：表示允许指定用户打开并修改文件，例如目录写表示允许从目录中删除或创建新的文件或目录。
- ☑ 执行权限（x）：表示允许指定用户将该文件作为一个程序执行，例如对目录表示允许你在目录中查找，并能用 cd 命令将工作目录切换到该目录。

Linux 系统在创建文件时，会自动把该文件的读写权限分配给其属主，使用户能够显示和修改该文件，也可以将这些权限改变为其他的组合形式。一个文件如果有执行权限，则允许它作为一个程序被执行。

2．内存管理

内存管理是计算机编程最基本的领域之一。在很多脚本语言中，我们不必担心内存是如何管理的，这并不能使得内存管理的重要性有一点点降低。对实际编程来说，理解内存管理器的能力与局限性至关重要。在大部分系统语言中必须进行内存管理，例如 C 和 C++。

追溯到在 Apple II 上进行汇编语言编程的时代，那时内存管理还不是个大问题。实际上在运行整个系统时，系统有多少内存我们就有多少内存。我们甚至不必费心思去弄明白它有多少内存，因为每一台机器的内存数量都相同。所以，如果内存需要非常固定，那么只需要选择一个内存范围并使用它即可。

但即使是在这样一个简单的计算机中也会有问题，尤其是当我们不知道程序的每个部分将需要多少内存时。如果我们的空间有限，而内存需求是变化的，那么需要用一些方法来满足下面的需求。

（1）确定是否有足够的内存来处理数据。

（2）从可用的内存中获取一部分内存。

（3）向可用内存池（pool）中返回部分内存，以使其可以由程序的其他部分或者其他程序使用。

实现上述需求的程序库称为分配程序（allocators），因为它们负责分配和回收内存。程序的动态性越强，内存管理就越重要，你的内存分配程序的选择也就更重要。让我们来了解可用于内存管理的不同方法、它们的好处与不足，以及它们最适用的情形。

3．进程管理

在 Linux 操作系统中包括了 3 种不同类型的进程，分别是交互进程、批处理进程和守护进程。每种进程都有自己的特点和属性。交互进程是由一个 Shell 启动的进程，既可以在前台运行，也可以在后台运行。批处理进程和终端没有联系，是一个进程序列。系统守护进程是 Linux 系统启动时启动的进程，并在后台运行。

Linux 管理进程的最好方法就是使用命令行下的系统命令。Linux 下面的进程涉及的命令有 ps、kill、pgrep 等工具。

（1）父进程和子进程

父进程和子进程的关系是管理和被管理的关系，当父进程终止时，子进程也随之而终止，但子进程终止，父进程并不一定终止。例如 httpd 服务器运行时，我们可以杀掉其子进程，父进程并不会因为子进程的终止而终止。在进程管理中，当我们发现占用资源过多或无法控制的进程时，应该杀死它，以保护系统的稳定安全运行。

（2）进程命令

在 Linux 中，通过命令来管理和操作进程，其中常用的命令可以分为如下几类。

① 监视进程命令

☑ Ps（process status 命令）：用于显示瞬间行程（process）的动态，其使用方式如下所示。

```
ps [options] [--help]
```

ps 的参数非常多，常用参数的具体说明如下。

- -A：列出所有的行程。
- -w：显示加宽可以显示较多的资讯。
- -au：显示较详细的资讯。
- -aux：显示所有包含其他使用者的行程。

☑ pstree 命令：功能是将所有行程以树状图显示，树状图将会以 pid（如果有指定）或是以 init 这个基本行程为根（root），如果有指定使用者 id，则树状图会只显示该使用者所拥有的行程。其使用方式如下所示。

```
pstree [-a] [-c] [-h|-Hpid] [-l] [-n] [-p] [-u] [-G|-U] [pid|user] pstree -V
```

常用参数的具体说明如下。

- -a：显示该行程的完整指令及参数，如果是被记忆体置换出去的行程则会加上括号。
- -c：如果有重复的行程名，则分开列出（预设值会在前面加上*）。

☑ rop 命令：用于实时显示 process 的动态，其使用方式如下所示。

```
top [-] [d delay] [q] [c] [S] [s] [i] [n] [b]
```

常用参数的具体说明如下。

- d：改变显示的更新速度，或是在交谈式指令列（interactive command）按 s。
- q：没有任何延迟的显示速度，如果使用者是有 superuser 的权限，则 top 将会以最高的优先序执行。
- c：切换显示模式，共有两种模式，一是只显示执行档的名称，另一种是显示完整的路径与名称。

- S：累积模式，会将已完成或消失的子行程（dead child process）的 CPU time 累积起来。
- s：安全模式，将交谈式指令取消，避免潜在的危机。
- i：不显示任何闲置（idle）或无用（zombie）的行程。
- n：更新的次数，完成后将会退出 top。
- b：批次档模式，搭配 n 参数一起使用，可以用来将 top 的结果输出到档案内。

② 控制进程命令

向 Linux 系统的内核发送一个系统操作信号和某个程序的进程标识号，系统内核就可以对进程标识号指定的进程进行操作。例如在 top 命令中会看到系统运行的许多进程，有时就需要使用 kill 中止某些进程来提高系统资源。在安装和登录命令中使用多个虚拟控制台的作用是，当一个程序出错造成系统死锁时可以切换到其他虚拟控制台工作关闭这个程序。此时使用的命令就是 kill，因为 kill 是大多数 Shell 内部命令可以直接调用的。

在 Linux 系统中，使用 kill 命令来控制进程。kill 可以删除执行中的程序或工作，可以将指定的信息送至程序，预设的信息为 SIGTERM(15)，可将指定程序终止。若仍无法终止该程序，可使用 SIGKILL(9)信息尝试强制删除程序。程序或工作的编号可利用 ps 指令或 jobs 指令查看。

在现实应用中，有如下两种使用 kill 命令的方式。

```
kill [-s <信息名称或编号>][程序]
kill [-l <信息编号>]
```

各个参数的具体说明如下。

- -s <信息名称或编号>：指定要送出的信息。
- -l <信息编号>：如果不加<信息编号>选项，则-l 参数会列出全部的信息名称。

注意： 进程是 Linux 系统中一个非常重要的概念。Linux 是一个多任务的操作系统，系统上经常同时运行着多个进程。我们不关心这些进程究竟是如何分配的，或者是内核如何管理分配时间片的，我们所关心的是如何去控制这些进程，让它们能够很好地为用户服务。

☑ 进程优先级设定（nice 命令）

使用 nice 命令可以使用更改过的优先顺序来执行程序，如果未指定程序则会显示出目前的进程优先顺序，默认的 adjustment 为 10，范围为−20（最高优先序）到 19（最低优先序）。使用 nice 命令的方式如下所示。

```
nice [-n adjustment] [-adjustment] [--adjustment=adjustment] [--help] [--version] [command [arg...]
```

各个参数的具体说明如下。

- -n adjustment，-adjustment，--adjustment=adjustment：都将该原有优先序的增加 adjustment。
- --help：显示求助讯息。
- --version：显示版本资讯。

4．设备驱动程序

设备驱动程序用于与系统连接的输入/输出装置通信，如硬盘、软驱、各种接口、声卡等。按照“万物皆文件（everything is a file）”的说法，对外设的访问可利用/dev 目录下的设备文件来完成，程序对设备的处理完全类似于常规的文件。设备驱动程序的任务在于支持应用程序经由设备文件与设备通信。通常可以将外设分为以下两类。

（1）字符设备：提供连续的数据流，应用程序可以顺序读取，通常不支持随机存取。相反，此类设备支持按字节/字符来读写数据。举例来说，调制解调器是典型的字符设备。

（2）块设备：应用程序可以随机访问设备数据，程序可自行确定读取数据的位置。硬盘是典型的块设

备，应用程序可以寻址磁盘上的任何位置，并由此读取数据。此外，数据的读写只能以块（通常是 512B）的倍数进行。与字符设备不同，块设备并不支持基于字符的寻址。

在现实开发应用中，编写块设备的驱动程序比字符设备要复杂得多，因为内核为提高系统性能广泛地使用了缓存机制。

5．网络

网卡也可以通过设备驱动程序控制，但在内核中属于特殊状况，因为网卡不能利用设备文件访问。原因在于在网络通信期间，数据打包到了各种协议层中。在接收到数据时，内核必须针对各协议层的处理，对数据进行拆包与分析，然后才能将有效数据传递给应用程序。在发送数据时，内核必须首先根据各个协议层的要求打包数据，然后才能发送。

为支持通过文件接口处理网络连接（按照应用程序的观点），Linux 使用了源于 BSD 的套接字抽象。套接字可以看作应用程序、文件接口、内核的网络实现之间的代理。

2.4　分析 Linux 内核源码

长期以来，学习内核的最好方法就是学习内核代码，因为内核代码本身就是最好的参考资料，其他任何经典或非经典的教科书都只是起辅助作用，不能也不应该取代内核代码在我们学习过程中的主导地位。本节将简要介绍 Linux 内核源码的基本知识。

2.4.1　源码目录结构

Linux 内核源码的官方下载地址为 http://www.kernel.org/，如图 2-7 所示。

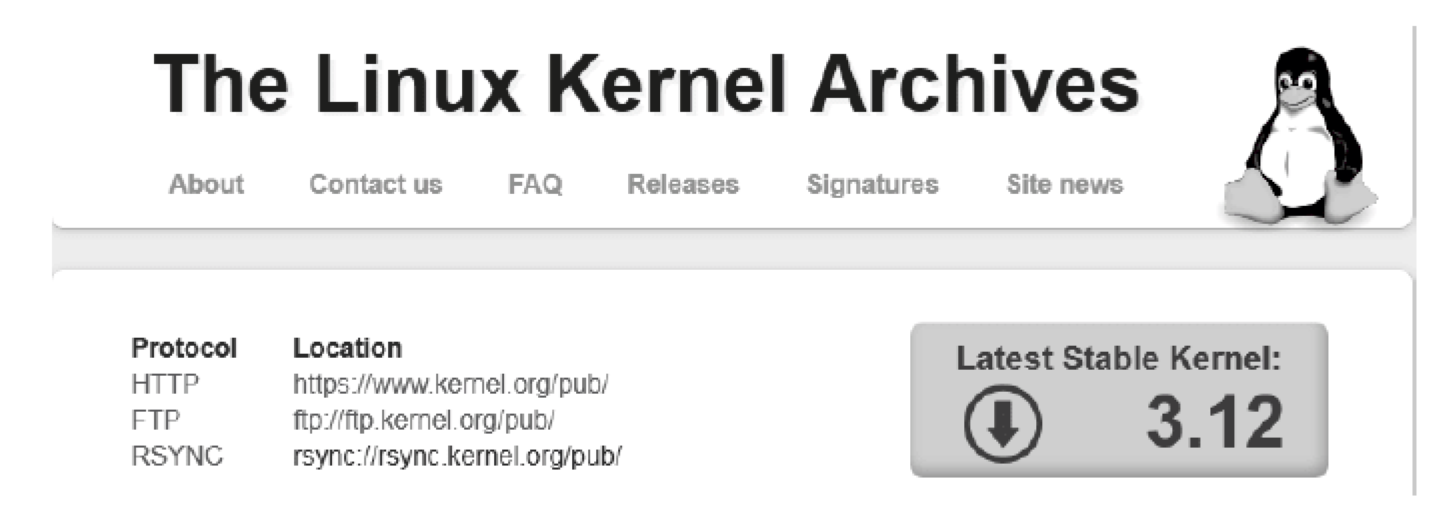


图 2-7　Linux 内核官方下载界面

当下载内核代码后，很有必要知道内核源码的整体分布情况。通常其内核代码保存在/usr/src/linux 目录下，该目录下的每一个子目录都代表了一个特定的内核功能性子集，接下来将针对 Linux 3.12 版本进行简单描述。

（1）目录 Documentation

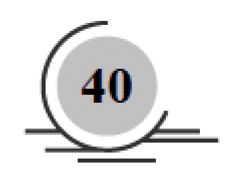
此目录下面没有内核代码，只有很多质量参差不齐的文档，但往往能够给我们提供很多的帮助。

（2）目录 arch

所有与体系结构相关的代码都在此目录以及“include/asm-*/”目录中，Linux 支持的每种体系结构在 arch 目录下都有对应的子目录，而在每个体系结构特有的子目录下又会至少包含如下 3 个子目录。

☑　kernel：存放支持体系结构特有的诸如信号量处理和 SMP 之类特征的实现。

☑　lib：存放体系结构特有的对诸如 strlen 和 memcpy 之类的通用函数的实现。



☑　mm：存放体系结构特有的内存管理程序的实现。

除了上述 3 个子目录之外，大多数体系结构在必要的情况下还有一个 boot 子目录，包含了在这种硬件平台上启动内核所使用的部分或全部平台特有代码。另外在大部分体系结构所特有的子目录中，还应该根据需要包含供附加特性使用的其他子目录。例如，i386 目录包含一个 math-emu 子目录，其中包括了在缺少浮点运算处理器（FPU）的 CPU 上运行模拟 FPU 的代码。

（3）目录 drivers

此目录是内核中最庞大的一个目录，显卡、网卡、SCSI 适配器、PCI 总线、USB 总线和其他任何 Linux 支持的外围设备或总线的驱动程序都可以在这里找到。

（4）目录 fs

在此目录中保存了虚拟文件系统（Virtual File System，VFS）的代码，还有各个不同文件系统的代码。Linux 支持的所有文件系统在 fs 目录下面都有一个对应的子目录。例如 ext2 文件系统对应的是 fs/ext2 目录。

一个文件系统是存储设备和需要访问存储设备的进程之间的媒介。存储设备可能是本地物理上可访问的，例如硬盘或 CD-ROM 驱动器，它们分别使用 ext2/ext3 和 isofs 文件系统，也可能是通过网络访问的，使用 NFS 文件系统。

还有一些虚拟文件系统，例如 proc 是以一个标准文件系统出现的，然而它其中的文件只存在于内存中，并不占用磁盘空间。

（5）目录 include

此目录中包含了内核中大部分的头文件，它们按照“include/asm-*/”的子目录格式进行分组。这种格式的子目录有多个，每一个都对应着一个 arch 的子目录，例如 include/asm-alpha、include/asm-arm 和 include/asm-i386 等。在每个子目录中的文件中，都定义了支持给定体系结构所必需的预处理器宏和内联函数，这些内联函数多数都是全部或部分使用汇编语言实现的。

在编译内核时，系统会建立一个从 include/asm 目录到目标体系结构特有的目录的符号链接。例如对于 ARM 平台，就是 include/asm-arm 到 include/asm 的符号链接。因此，体系结构无关部分的内核代码可以使用如下形式包含体系相关部分的头文件。

（6）目录 init

此目录保存了内核的初始化代码，包括 main.c、创建早期用户空间的代码以及其他初始化代码。

（7）目录 ipc

IPC 即进程间通信（Inter Process Communication），在此目录中包含了共享内存、信号量以及其他形式 IPC 的代码。

（8）目录 kernel

此目录是内核中最核心的部分，包括进程的调度（kernel/sched.c），以及进程的创建和撤销（kernel/ fork.c 和 kernel/exit.c）等，和平台相关的另外一部分核心的代码在 arch/*/kernel 目录下。

（9）目录 lib

此目录中保存了库代码，这些代码实现了一个标准 C 库的通用子集，包括字符串和内存操作的函数（strlen、mmcpy 和其他类似的函数）以及有关 sprintf 和 atoi 的系列函数。与 arch/lib 下的代码不同，这里的库代码都是使用 C 编写的，在内核新的移植版本中可以直接使用。

（10）目录 mm

此目录中包含了体系结构无关部分的内存管理代码，体系相关的部分位于 arch/*/mm 目录下。

（11）目录 net

此目录中保存了和网络相关的代码，实现了各种常见的网络协议，如 TCP/IP、IPX 等。

（12）目录 scripts

该目录下没有内核代码，只包含用来配置内核的脚本文件。当运行 make menuconfig 或者 make xconfig 之类的命令配置内核时，用户就是和位于这个目录下的脚本进行交互的。

（13）目录 block

此目录中保存了 block 层的实现代码，最初 block 层的代码一部分位于 drivers 目录下，一部分位于 fs 目录下，从 2.6.15 开始，block 层的核心代码被提取出来放在了顶层的 block 目录下。

（14）目录 crypto

此目录中保存了内核本身所用的加密 API 信息，实现了常用的加密和散列算法，还有一些压缩和 CRC 校验算法。

（15）目录 security

此目录下包括不同的 Linux 安全模型的代码，例如 NSA Security-Enhanced Linux。

（16）目录 sound

在此目录下保存了声卡驱动以及其他声音相关的代码。

（17）目录 usr

此目录实现了用于打包和压缩的 cpio 等。

2.4.2 浏览源码的工具

俗话说“工欲善其事，必先利其器”，使用一个功能强大并方便的代码浏览工具有助于我们学习内核代码，下面将简单介绍浏览 Linux 内核源码的常用工具。

1．Source Insight

Windows 下最方便快捷的代码浏览工具是 Source Insight，这是一款商业软件。这是一个面向项目开发的程序编辑器和代码浏览器，它拥有内置的对 C/C++、C#和 Java 等程序的分析。Source Insight 可以分析源代码并动态维护它自己的符号数据库，并自动为你显示有用的上下文信息。Source Insight 不仅仅是一个强大的程序编辑器，它还能显示 reference trees、class inheritance diagrams 和 call trees。Source Insight 提供了最快速的对源代码的导航和任何程序编辑器的源信息。Source Insight 提供了快速和革新的访问源代码和源信息的能力。与众多其他编辑器产品不同，Source Insight 能在你编辑的同时分析你的源代码，为你提供实用的信息并立即进行分析。

安装 Source Insight 后，需要先打开 Source Insight 并创建一个工程，然后将内核代码加入到该工程中，并进行文件同步，这样就可以在代码之间进行关联阅读了。

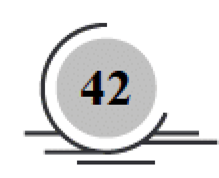
Source Insight 的缺点是没有对应 Linux 的版本。因此对于很多 Linux 初学者来说，在一个完全的 Linux 环境下进行学习，需要寻找一个可以取代 Source Insight 的代码浏览工具。

2．Vim+Cscope

Linux 环境下的最佳浏览工具是 Vim，各种 Linux 发行版都会默认进行安装。虽然 Vim 默认的编辑界面很普通甚至说丑陋，但是可以通过配置文件.vimrc 添加不同的界面效果。同时还可以配合 TagList、WinManager 等很多好用的插件或工具，将 Vim 打造成一个不次于 Source Insight 的代码浏览编辑工具。

3．LXR

LXR（Linux Cross Reference）也是一种比较流行的 Linux 内核源代码浏览工具，其下载地址为 http://lxr.linux.no/。



如果只是浏览 Linux 内核代码，则并不需要安装 LXR，因为在网站 http://lxr.linux.no/上已经提供了几乎所有版本的 Linux 内核代码，只需要登录该网站，选择某一特定的内核版本后就可以在内核代码之间进行关联阅读。

当登录网站并选择内核版本后，在查找框内输入要查找的内核代码符号名称，然后就可以搜索得到所有以超链接形式给出的对该符号定义和引用的确切位置。

注意：为什么用汇编语言编写内核代码

很多读者可能要问，Java、C++和 C#功能强大，Visual Basic 易于使用，但是为什么还要使用古老的汇编语言来编写内核代码呢？这是因为以下 3 个方面的考虑。

（1）Linux 内核中的底层代码直接和硬件打交道，需要一些专用的指令，而这些指令在 C 语言中并无对应的语言成分。

（2）内核中实现某些操作的过程、代码段或函数，在运行时会很频繁地被调用，这时用汇编语言编写，其时间效率会有大幅度提高。

（3）在某些特别的场合，一段代码的空间效率也很重要，例如操作系统的引导程序一定要容纳在磁盘的第一个扇区中，多一个字节都不行。这时只能用汇编语言编写。

在 Linux 内核代码中，以汇编语言编写的代码有如下两种不同的形式。

☑ 完全的汇编代码，这样的代码采用“.s”作为文档名的后缀。

☑ 嵌入在 C 语言代码中的汇编语言片段。

对于新接触 Linux 内核源码的读者，即使比较熟悉 i386 汇编语言，在理解内核中的汇编代码时都会感到困难。原因是在内核的汇编代码中，采用的是不同于常用 Intel i386 汇编语言的 AT&T 格式的汇编语言，而在嵌入 C 语言代码的汇编语言片段中，更是增加了一些指导汇编工具如何分配使用寄存器、如何与 C 语言代码中定义的变量相结合的语言成分。这些成分使得嵌入 C 语言代码的汇编语言片断实际上变成了一种介于汇编和 C 语言之间的一种中间语言。

2.4.3 GCC 特性

Linux 内核使用 GCC（GNU Compiler Collection）套件的几个特殊功能。这些功能包括提供快捷方式和简化以及向编译器提供优化提示等。GCC 和 Linux 是出色的组合。尽管它们是独立的软件，但是 Linux 完全依靠 GCC 在新的体系结构上运行。Linux 还利用 GCC 中的特性（称为扩展）实现更多的功能和优化。

（1）基本功能

概括来说，GCC 具有如下两大功能。

☑ 功能性：扩展提供新功能。

☑ 优化：扩展帮助生成更高效的代码。

GCC 允许通过变量的引用识别类型，这种操作支持泛型编程。在 C++、Ada 和 Java 语言等许多现代编程语言中都可以找到相似的功能。例如 Linux 使用 typeof 构建 min 和 max 等依赖于类型的操作。使用 typeof 构建一个泛型宏的代码如下所示。

```
#define min(x, y) ({
	typeof(x) _min1 = (x);
	typeof(y) _min2 = (y);
	(void) (&_min1 == &_min2);
	_min1 < _min2 ? _min1 : _min2; })
```

GCC 还支持范围，在 C 语言的许多方面都可以使用范围。最常见的是在 switch/case 块中的 case 语句中使用。使用 switch/case 也可以通过使用跳转表实现进行编译器优化。在复杂的条件结构中，通常依靠嵌套

的 if 语句实现与下面代码相同的结果，但是下面的代码更简洁，具体代码如下所示。

```
static int sd_major(int major_idx){
        switch (major_idx) {
        case 0:
                return SCSI_DISK0_MAJOR;
        case 1 ... 7:
                return SCSI_DISK1_MAJOR + major_idx - 1;
        case 8 ... 15:
                return SCSI_DISK8_MAJOR + major_idx - 8;
        default:
                BUG();
                return 0;   /* shut up gcc */
        }
}
```

（2）属性

GCC 允许声明函数、变量和类型的特殊属性，以便指示编译器进行特定方面的优化和更仔细的代码检查。使用方式非常简单，只需在声明后面加上如下代码即可。

```
attribute__ (( ATTRIBUTE ))
```

其中 ATTRIBUTE 是属性的说明，多个说明之间以逗号分隔。GCC 可以支持十几个属性，接下来将介绍一些比较常用的属性。

- ☑ 属性 noreturn：用在函数中，表示该函数从不返回。它能够让编译器生成较为优化的代码，消除不必要的警告信息。
- ☑ 属性 format(archetype, string-index, first-to-check)：用在函数中，表示该函数使用 printf、scanf、strftime 或 strfmon 风格的参数，并可以让编译器检查函数声明和函数实际调用参数之间的格式化字符串是否匹配。
  - ➢ archetype：指定是哪种风格。
  - ➢ string-index：指定传入函数的第几个参数是格式化字符串。
  - ➢ first-to-check：指定从函数的第几个参数开始按照上述规则进行检查。

例如下面的内核代码。

```
++++ include/linux/kernel.h
 static inline int printk(const char *s, ...)
        __attribute__ ((format (printf, 1, 2)));
```

这表示 printk 的第一个参数是格式化字符串，从第二个参数开始根据格式化字符串检查参数。

- ☑ 属性 unused：用于函数和变量，表示该函数或变量可能并不使用，这个属性能够避免编译器产生警告信息。
- ☑ 属性 aligned(ALIGNMENT)：常用在变量、结构或联合中，用于设定一个指定大小的对齐格式，以字节为单位，例如下面的内核代码。

```
++++ drivers/usb/host/ohci.h
 struct ohci_hcca {
 #define NUM_INTS 32
        __hc32   int_table [NUM_INTS];     /* periodic schedule */
        /*
         * OHCI defines u16 frame_no, followed by u16 zero pad.
         * Since some processors can't do 16 bit bus accesses,
         * portable access must be a 32 bits wide.
         */
```

```
        __hc32   frame_no;          /* current frame number */
        __hc32   done_head;         /* info returned for an interrupt */
        u8   reserved_for_hc [116];
        u8   what [4];          /* spec only identifies 252 bytes :) */
 } __attribute__ ((aligned(256)));
```

上述代码表示结构体 ohci_hcca 的成员以 256 字节对齐。如果 aligned 后面不紧跟一个指定的数字值，那么编译器将依据目标机器情况使用最大、最有益的对齐方式。

需要注意的是，attribute 属性的效果与连接器也有关，如果连接器最大只支持 16 字节对齐，那么此时定义 32 字节对齐也是无济于事的。

☑ 属性 packed：用在变量和类型中，当用在变量或结构体成员时表示使用最小可能的对齐，当用在枚举、结构体或联合类型时表示该类型使用最小的内存。属性 packed 多用于定义硬件相关的结构时，使元素之间不会因对齐产生问题，例如下面的内核代码。

```
++++ include/asm-i386/desc.h
 struct usb_interface_descriptor {
        __u8   bLength;
        __u8   bDescriptorType;
        __u8   bInterfaceNumber;
        __u8   bAlternateSetting;
        __u8   bNumEndpoints;
        __u8   bInterfaceClass;
        __u8   bInterfaceSubClass;
        __u8   bInterfaceProtocol;
        __u8   iInterface;
 } __attribute__ ((packed));
```

在上述代码中，__attribute__((packed))告诉编译器 usb_interface_descriptor 的元素为 1 字节对齐，不要再添加填充位。因为定义此结构的代码和 usb spec 中的是完全一样的。如果不给编译器这个暗示，则编译器会依据平台的类型在结构的每个元素之间添加一定的填充位，使用这个结构时就不能达到预期的结果。

（3）内建函数

在 GCC 中提供了大量的内建函数，其中有很多是标准 C 库函数的内建版本，例如 memcpy()，它们的功能与对应的 C 库函数的功能相同，在此不再进行讲解。

在内建函数中，还有很多函数的名字是以__builtin 开始的，接下来将对__builtin_expect()进行详细分析，其他__builtin_xxx()函数的原理和此函数类似，本书中将不再一一介绍。

函数__builtin_expect()的格式如下所示。

```
__builtin_expect (long exp, long c)
```

为什么 Linux 会推出__builtin_xxx()函数呢？这是因为大部分代码在分支预测方面做的比较糟糕，所以 GCC 提供了此内建函数来帮助处理分支预测并优化程序。

☑ 第一个参数 exp：是一个整型的表达式，返回值也是此 exp。

☑ 第二个参数 c：其值必须是一个编译期的常量。

由此可见，此内建函数的意思就是 exp 的预期值为 c，编译器可以根据这个信息适当地重排条件语句块的顺序，将符合这个条件的分支放在合适的地方。

下面看此函数在 Linux 内核中的应用，具体代码如下所示。

```
++++ include/linux/compiler.h
#define likely(x)       __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)
```

unlikely(x)用于告诉编译器条件 x 发生的可能性不大，likely 用于告诉编译器条件 x 发生的可能性很大。

它们一般用在条件语句中，if 语句不变，当 if 条件为 1 可能性非常小时，可以在条件表达式外面包装一个 unlikely()，那么这个条件块中语句的目标码可能就会被放在一个比较远的位置，以保证经常执行的目标码更紧凑。如果可能性非常大，则使用 likely()包装。

2.4.4 链表的重要性

链表和本书讲解的驱动密切相关，例如 USB 驱动。鉴于链表在内核中的特殊地位，有必要在此对其做一番陈述。内核中链表的实现位于 include/linux/list.h 文件中，链表数据结构的定义也很简单，具体代码如下所示。

```
struct list_head {
        struct list_head *next, *prev;
};
```

结构 list_head 包含两个指向 list_head 结构的指针 prev 和 next，由此可见，内核中的链表实际上都是双链表。学习过 C 语言的读者应该知道，链表的定义结构如下所示。

```
struct list_node {
      struct list_node *next;
      ElemType       data;
};
```

通过上述格式使用链表，对于每一种数据类型都要定义它们各自的链表结构。而内核中的链表却与此不同，它并没有数据域，不是在链表结构中包含数据，而是在描述数据类型的结构中包含链表。

如果在 hub 驱动中使用 struct usb_hub 来描述 hub 设备，hub 需要处理一系列的事件，例如当探测到一个设备连进来时，就会执行一些代码去初始化该设备，所以 hub 就创建了一个链表来处理各种事件，这个链表的结构如图 2-8 所示。

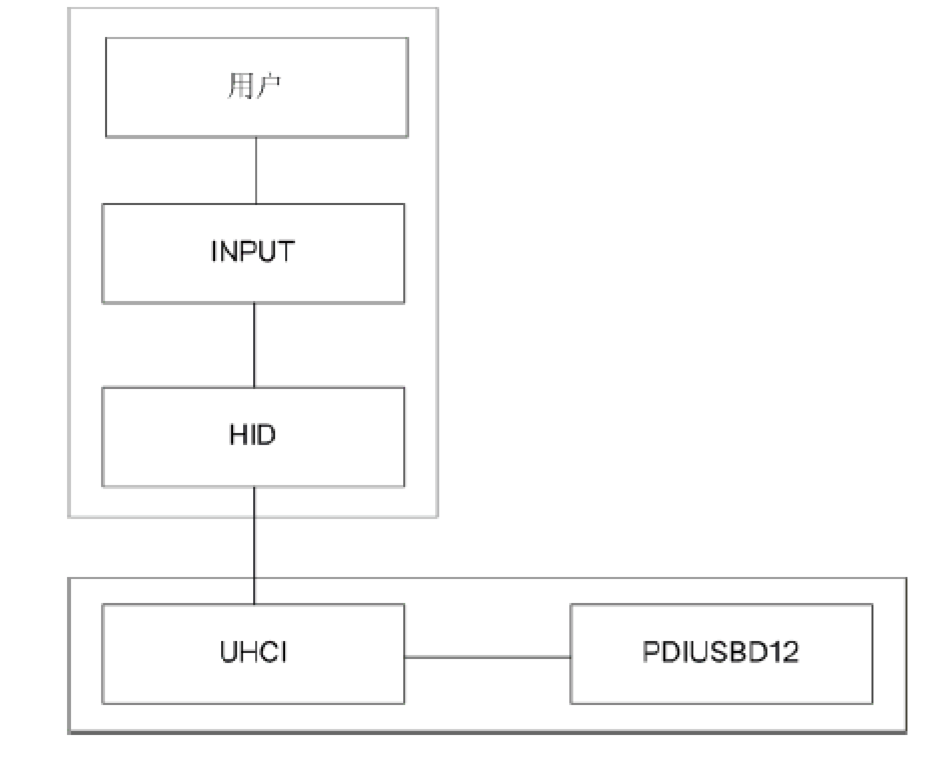


图 2-8 hub 链表的结构

在 Linux 代码中，完整展示了链表的操作过程，接下来将简单剖析对应的 Linux 代码。

（1）声明与初始化

可以使用如下两种方式来声明链表。

- ☑ 使用 LIST_HEAD 宏在编译时静态初始化。
- ☑ 使用 INIT_LIST_HEAD()在运行时进行初始化。

对应的 Linux 代码如下所示。

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)

static inline void INIT_LIST_HEAD(struct list_head *list)
{
        list->next = list;
        list->prev = list;
}
```

无论采用哪种方式，新生成的链表头的两个指针 next、prev 都初始化为指向自己。

（2）判断链表是否为空

对应的 Linux 代码如下所示。

```
static inline int list_empty(const struct list_head *head)
{
	return head->next == head;
}
```

（3）插入操作

建立链表后，就不可避免地要对其进行操作，例如向里面添加数据。使用函数 list_add()和 list_add_ tail()可以完成添加数据的工作。对应的 Linux 代码如下所示。

```
static inline void list_add(struct list_head
	*new, struct list_head *head)
{
	__list_add(new, head, head->next);
}
static inline void list_add_tail(struct
	list_head *new, struct list_head *head)
{
	__list_add(new, head->prev, head);
}
```

其中，函数 list_add()将数据插入在 head 之后，函数 list_add_tail()将数据插入在 head->prev 之后。对于循环链表来说，因为表头的 next、prev 分别指向链表中的第一个和最后一个节点，所以函数 list_add()和 list_add_tail()的区别并不大。

（4）删除操作

可以使用函数 list_replace_init()从链表中删除一个元素，并且将其初始化。对应的 Linux 代码如下所示。

```
static inline void list_replace_init(struct list_head *old,
					struct list_head *new)
{
	list_replace(old, new);
	INIT_LIST_HEAD(old);
}
```

（5）遍历操作

在内核中的链表仅保存了 list_head 结构的地址，我们应该如何通过它获取一个链表节点真正的数据项呢？此时就需要使用 list_entry 宏，通过它可以很容易地获得一个链表节点的数据。对应的 Linux 代码如下所示。

```
#define list_entry(ptr, type, member) \
	container_of(ptr, type, member)
```

假如以 hub 驱动为例，当要处理 hub 的事件时，需要知道具体是哪个 hub 触发了这起事件。而 list_ entry 的作用是从 struct list_head event_list 中得到它所对应的 struct usb_hub 结构体变量。例如下面的代码。

```
struct list_head *tmp;
struct usb_hub *hub;
tmp = hub_event_list.next;
hub = list_entry(tmp, struct usb_hub, event_list);
```

通过上述代码，从全局链表 hub_event_list 中取出一个叫做 tmp 的结构体变量，然后通过 tmp 获得它所对应的 struct usb_hub。

2.4.5　Kconfig 和 Makefile

Kconfig 和 Makefile 是浏览内核代码时最为依赖的两个文件之一。几乎 Linux 内核中的每一个目录下都有一个 Kconfig 文件和一个 Makefile 文件。通过 Kconfig 和 Makefile，可以让我们了解一个内核目录下的结构。在研究内核的某个子系统、某个驱动或其他某个部分之前，需要仔细阅读目录下对应的 Kconfig 和 Makefile 文件。

（1）Kconfig 结构

每种平台对应的目录下都有一个 Kconfig 文件，例如 arch/i386/Kconfig。Kconfig 文件通过 source 语句可以构建一个 Kconfig 树。文件 arch/i386/Kconfig 的代码片段如下所示。

```
mainmenu "Linux Kernel Configuration"

config X86_32
	bool
	default y
	help
	  This is Linux's home port.   Linux was
originally native to the Intel
	  386, and runs on all the later x86
processors including the Intel
	  486, 586, Pentiums, and various
instruction-set-compatible chips by
	  AMD, Cyrix, and others.
……
source "init/Kconfig"

menu "Processor type and features"

source "kernel/time/Kconfig"
……
config KTIME_SCALAR
	bool
	default y
```

Kconfig 的详细语法规则可以参看内核文档 Documentation/kbuild /kconfig-language.txt，下面对其进行简单介绍。

☑　菜单项：关键字 config 可以定义一个新的菜单项，例如下面的代码。

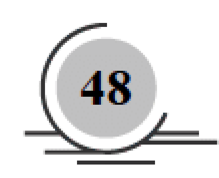
```
config MODVERSIONS
	bool "Set version information on all module symbols"
	depends on MODULES
	help
	  Usually, modules have to be recompiled whenever you switch to a new
	  kernel.   ...
```

后面的代码定义了该菜单项的属性，包括类型、依赖关系、选择提示、帮助信息和默认值等。

常用的类型有 bool、tristate、string、hex 和 int。bool 类型只能被选中或不选中，tristate 类型菜单项多了编译成内核模块的选项。

依赖关系是通过 depends on 或 requires 定义的，指出此菜单项是否依赖于另外一个菜单项。

帮助信息需要使用 help 或---help---来指出。



☑ 菜单组织结构：菜单选项通过两种方式来组成树状结构，具体说明如下。

➢ 第一种方式：使用关键字 menu 显式声明为菜单，例如下面的代码。

```
menu "Bus options (PCI, PCMCIA, EISA, MCA, ISA)"
 config PCI
...
 endmenu
```

➢ 第二种方式：也可以使用依赖关系确定菜单结构，例如下面的代码。

```
config MODULES
    bool "Enable loadable module support"

config MODVERSIONS
    bool "Set version information on all module symbols"
    depends on MODULES

comment "module support disabled"
    depends on !MODULES
```

其中菜单项 MODVERSIONS 依赖于 MODULES，所以它就是一个子菜单项。这要求菜单项和它的子菜单项同步显示或不显示。

☑ 关键字 Kconfig：Kconfig 文件描述了一系列的菜单选项，除帮助信息外，文件中的每一行都以一个关键字开始，主要有 config、menuconfig、choice/endchoice、comments、menu/endmenu、if/endif、source 等，只有前 5 个可以用在菜单项定义的开始，它们都可以结束一个菜单项。

（2）Makefile

Linux 内核的 Makefile 分为如下 5 个组成部分。

☑ Makefile：最顶层的 Makefile。

☑ .config：内核的当前配置文档，编译时成为顶层 Makefile 的一部分。

☑ arch/$(ARCH)/Makefile：和体系结构相关的 Makefile。

☑ Makefile.*：一些特定 Makefile 的规则。

☑ kbuild 级别 Makefile：各级目录下大概约 500 个文档，编译时根据上层 Makefile 传下来的宏定义和其他编译规则，将源代码编译成模块或编入内核。顶层的 Makefile 文档读取.config 文档的内容，并总体上负责 build 内核和模块。Arch Makefile 则提供补充体系结构相关的信息。其中.config 的内容是在 make menuconfig 时，通过 Kconfig 文档配置的结果。

假如想把自己写的一个 Flash 的驱动程序加载到工程中，并且能够通过 menuconfig 配置内核时会选择该驱动，此时该怎么办呢？其实借助 Makefile 就可以实现，解决流程如下。

☑ 将编写的 flashtest.c 文档添加到/driver/mtd/maps/目录下。

☑ 修改/driver/mtd/maps 目录下的 kconfig 文档，修改代码如下所示。

```
config MTD_flashtest
tristate "ap71 flash"
```

这样当运行 make menuconfig 时会出现 ap71 flash 选项。

☑ 修改该目录下 makefile 文档，添加下面的代码内容。

```
obj-$(CONFIG_MTD_flashtest) += flashtest.o
```

此时当运行 make menuconfig 时会发现 ap71 flash 选项，假如选择了此选项，该选择就会保存在.config 文档中。当编译内核时会读取.config 文档，当发现 ap71 flash 选项为 yes 时，系统在调用/driver/mtd/maps/下的 makefile 时，会把 flashtest.o 加入到内核中。

2.5 学习 Linux 内核的方法

学习 Linux 内核的最大工作就是对内核代码进行分析，如果抱着走马观花、得过且过的态度，最终结果很可能是没有多大的收获。学习内核应该遵循科学、严谨的态度，要做到真正理解每一段代码的实现，并且在学习过程中要多问、多想、多记。上述学习 Linux 内核的方法非常重要，本节将通过两个具体的应用来演示学习 Linux 内核的方法。

2.5.1 分析 USB 子系统的代码

Linux 内核中 USB 子系统的代码位于 drivers/usb 目录下，进入该目录，执行命令 ls 后将会显示如下结果。

```
atm   class   core   gadget   host   image   misc   mon   serial   storage Kconfig
Makefile   README usb-skeleton.c
```

目录 drivers/usb 一共包含 10 个子目录和 4 个文件，为了理解每个子目录的作用，有必要首先阅读 README 文件。根据 README 文件的描述，了解 drivers/usb 目录下各个子目录的作用，具体说明如下。

（1）core

core 是内核开发者针对部分核心的功能特意编写的代码，用于为其他的设备驱动程序提供服务，例如申请内存，实现一些所有的设备都会需要的公共函数，并命名为 USB core。

（2）host

早期的内核结构并不像现在这般富有层次感，几乎所有的文件都直接堆砌在 drivers/usb/目录下，这其中包括 usb core 和其他各种设备驱动程序的代码。

后来在 drivers/usb/目录下单独列出了 core 子目录，用于存放一些比较核心的代码，例如整个 USB 子系统的初始化、root hub 的初始化、host controller 的初始化代码。

后来随着技术的发展，出现了多种 USB host controller，于是内核开发者把 host controller 有关的公共代码保留在 core 目录下，而其他各种 host controller 对应的特定代码则移到 host 目录下，让相应的负责人去维护。为此针对 host controller 单独创建了子目录 host，它用于存放与其相关的代码。

（3）gadget

用于存放 USB gadget 的驱动程序，控制外围设备如何作为一个 USB 设备和主机通信。例如，嵌入式开发板通常会支持 SD 卡，使用 USB 连接线将开发板连接到 PC 时，通过 USB gadget 架构的驱动，可以将该 SD 卡模拟成 U 盘。

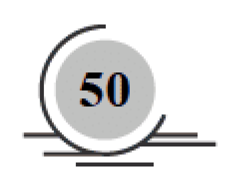
除 core、host 和 gadget 之外，其他几个目录分门别类存放了各种 USB 设备的驱动，例如 U 盘的驱动位于 storage 子目录，触摸屏和 USB 键盘鼠标的驱动位于 input 子目录。

因为我们的目的是研究内核对 USB 子系统的实现，而不是特定设备或 host controller 的驱动，所以通过对 README 文件的分析，应该进一步关注 core 子目录。

2.5.2 分析 USB 系统的初始化代码

通过分析 Kconfig 和 Makefile 文件，可以帮助我们在庞大复杂的内核代码中定位以及缩小目标代码的范围。为了研究内核对 USB 子系统的实现，需要在目标代码中找到 USB 子系统的初始化代码。

Linux 内核针对某个子系统或某个驱动，使用 subsys_initcall 或 module_init 宏来指定初始化函数。在内



核文件 drivers/usb/core/usb.c 中，可以发现下面的代码。

```
subsys_initcall(usb_init);
module_exit(usb_exit);
```

在上述代码中，可以将 subsys_initcall 理解为 module_init，只不过因为该部分代码比较核心，开发者们把它看作一个子系统，而不仅仅是一个模块。在 Linux 中，类似此类别的设备驱动被归结为一个子系统，例如 PCI 子系统和 SCSI 子系统。通常 drivers/目录下第一层的每个目录代表一个子系统，因为它们分别代表了一类设备。

subsys_initcall(usb_init)表示，函数 usb_init()是 USB 子系统的初始化函数，而 module_exit 则表示，usb_exit 函数是 USB 子系统结束时的清理函数。为了研究 USB 子系统在内核中的实现，需要从函数 usb_init()开始分析，对应的内核代码如下所示。

```
 static int __init usb_init(void)
 {
        int retval;
        if (nousb) {
            pr_info("%s: USB support
disabled\n", usbcore_name);
            return 0;
        }
        retval = ksuspend_usb_init();
        if (retval)
            goto out;
        retval = bus_register(&usb_bus_type);
        if (retval)
            goto bus_register_failed;
    retval = usb_host_init();
        if (retval)
            goto host_init_failed;
        retval = usb_major_init();
        if (retval)
            goto major_init_failed;
    retval = usb_register(&usbfs_driver);
        if (retval)
            goto driver_register_failed;
        retval = usb_devio_init();
        if (retval)
            goto usb_devio_init_failed;
        retval = usbfs_init();
        if (retval)
            goto fs_init_failed;
        retval = usb_hub_init();
    if (retval)
            goto hub_init_failed;
    retval = usb_register_device_driver
(&usb_generic_driver, THIS_MODULE);
        if (!retval)
            goto out;
        usb_hub_cleanup();
```

```
        hub_init_failed:
                usbfs_cleanup();
fs_init_failed:
                usb_devio_cleanup();
usb_devio_init_failed:
                usb_deregister(&usbfs_driver);
driver_register_failed:
                usb_major_cleanup();
major_init_failed:
        usb_host_cleanup();
host_init_failed:
                bus_unregister(&usb_bus_type);
bus_register_failed:
                ksuspend_usb_cleanup();
out:
                return retval;
}
```

接下来开始分析上述代码。

（1）标记__init

关于 usb_init，第一个问题是上述第一行代码中的__init 标记有什么意义？在前面讲解 GCC 扩展的特殊属性 section 时曾经提到，__init 修饰的所有代码都会被放在.init.text 节，当初始化结束后就可以释放这部分内存。但内核是如何调用到__init 所修饰的这些初始化函数的呢？为了回答这个问题，需要用到 subsys_initcall 宏的知识，它在文件 include/linux/init.h 中的定义格式如下所示。

```
#define subsys_initcall(fn)        __define_initcall("4",fn,4)
```

此时出现了一个新的宏__define_initcall，它用来将指定的函数指针 fn 存放到.initcall.init 节。对于 subsys_initcall 宏，则表示把 fn 存放到.initcall.init 的子节.initcall4.init。

为了理解.initcall.init、.init.text 和.initcall4.init 之类的符号，还需要了解和内核可执行文件相关的概念。内核可执行文件由许多链接在一起的对象文件组成。对象文件有许多节，如文本、数据、init 数据、bass 等。这些对象文件都是由一个称为链接器脚本的文件链接并装入的。这个链接器脚本的功能是将输入对象文件的各节映射到输出文件中。换句话说，它将所有输入对象文件都链接到单一的可执行文件中，将该可执行文件的各节装入指定地址处。vmlinux.lds 是保存在 arch/<target>/目录中的内核链接器脚本，它负责链接内核的各个节并将它们装入内存中特定偏移量处。

打开文件 arch/i386/kernel/vmlinux.lds，搜索关键字 initcall.init 后便会看到如下结果。

```
__inicall_start = .;
.initcall.init : AT(ADDR(.initcall.init) - 0xC0000000) {
*(.initcall1.init)
*(.initcall2.init)
*(.initcall1.init)
*(.initcall4.init)
*(.initcall5.init)
*(.initcall6.init)
*(.initcall7.init)
}
__initcall_end = .;
```

其中__initcall_start 指向.initcall.init 节的开始，__initcall_end 指向.initcall.init 节的结尾。而.initcall.init

节又被分为了如下 7 个子节。

```
.initcall1.init
.initcall2.init
.initcall1.init
.initcall4.init
.initcall5.init
.initcall6.init
.initcall7.init
```

宏 subsys_initcall 将指定的函数指针放在了.initcall4.init 子节，至于其他宏的功能也类似，例如 core_initcall 将函数指针放在了.initcall1.init 子节，device_initcall 将函数指针放在了.initcall6.init 子节等。

各个子节的顺序是确定的，即先调用.initcall1.init 中的函数指针，然后再调用.initcall2.init 中的函数指针。__init 修饰的初始化函数在内核初始化过程中调用的顺序和.initcall.init 节中函数指针的顺序有关，不同的初始化函数被放在不同的子节中，因此也就决定了它们的调用顺序。

（2）模块参数

在前面 usb_init 函数代码中，代码 nousb 在 drivers/usb/core/usb.c 文件中定义为如下格式。

```
static int nousb;     /* Disable USB when built into kernel image */
module_param_named(autosuspend, usb_autosuspend_delay, int, 0644);
MODULE_PARM_DESC(autosuspend, "default autosuspend delay");
```

从中可知 nousb 是个模块参数，用于在内核启动时禁止 USB 子系统。关于模块参数，可以在加载模块时指定，但是如何在内核启动时指定？打开系统的 grub 文件，然后找到 kernel 行，例如下面的代码。

```
kernel          /boot/vmlinuz-2.6.18-kdb root=/dev/sda1 ro splash=silent vga=0x314
```

其中的 root、splash、vga 等都表示内核参数。当某一模块被编译进内核时，它的模块参数便需要在 kernel 行来指定，其格式为：

```
模块 名.参数=值
```

例如下面的代码。

```
modprobe usbcore autosuspend=2
```

对应到 kernel 行的代码如下所示。

```
usbcore.autosuspend=2
```

通过命令 modinfo -p ${modulename}可以得知一个模块有哪些参数可以使用。而对于已经加载到内核中的模块，其模块参数会列举在/sys/module/$ {modulename}/parameters/目录下面，可以使用如下命令去修改。

```
echo -n ${value} > /sys/module/$ {modulename}/parameters/${parm}
```

关于函数 usb_init()，除了上面介绍的代码外，余下的代码分别完成 usb 各部分的初始化，其他代码的具体分析工作可以参阅下载 Linux 内核代码，具体含义可以参阅相关的书籍和资料。因为篇幅限制，此处不再进行详细介绍。

注意：学习内核是一项浩大的工程，在学习之前需要做到以下 3 个方面。

（1）熟练使用 Linux 操作系统

Linux 操作系统是 Linux 内核在用户层面的具体体现，只有熟练掌握 Linux 的基本操作，才能在内核学习的过程中达到事半功倍的效果。

（2）掌握操作系统理论基础

要掌握操作系统中比较基础的理论，例如分时（time-shared）和实时（real-time）的区别，进程的概念、CPU 和系统总线、内存的关系等。

（3）掌握 C 语言基础

不需要很精通 C 语言，但必须能够理解链表、散列表等数据结构的 C 实现，并熟练运用 GCC 编译器。总之对 C 语言越熟悉就会对我们的内核学习有帮助。

2.6　Linux 中的 3 类驱动程序

在 Linux 系统中主要有 3 类设备驱动，分别是字符设备驱动、块设备驱动和网络接口驱动，本节将简要讲解上述 3 类设备驱动的基本知识。

2.6.1　字符设备驱动

字符设备是指在 I/O 传输过程中以字符为单位进行传输的设备，例如键盘、打印机等。在此需要注意的是，以字符为单位并不一定意味着是以字节为单位，因为有的编码规则规定，1 个字符占 16 比特，合两个字节。字符设备驱动程序的结构如图 2-9 所示。

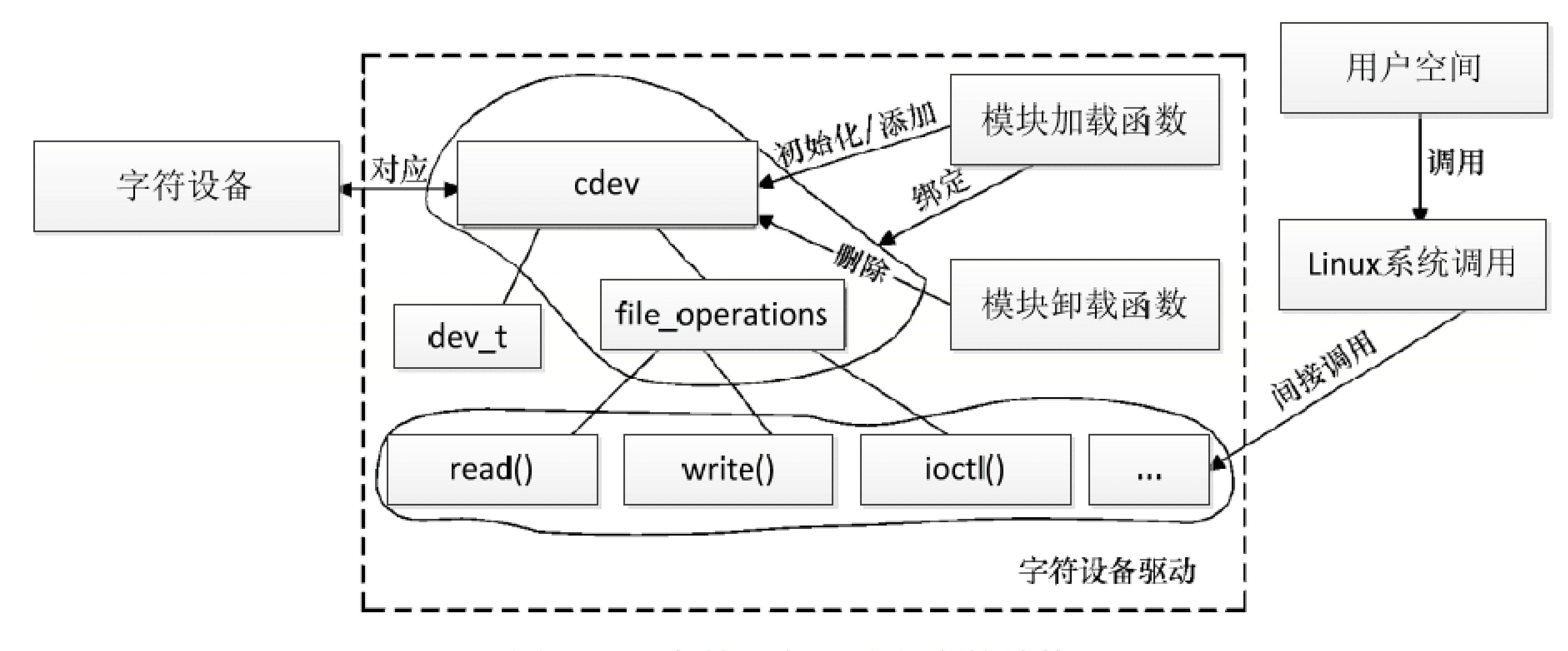


图 2-9　字符设备驱动程序的结构

在 Linux 系统中，字符设备以特别文件方式在文件目录树中占据位置并拥有相应的 i 节点。在 i 节点中的文件类型指明该文件是字符设备文件。可以使用与普通文件相同的文件操作命令对字符设备文件进行操作，例如打开、关闭、读、写等。概括来说，字符设备驱动主要做如下 3 件事。

- ☑ 定义一个结构体 static struct file_operations 变量，在里面定义一些设备的打开、关闭、读、写、控制函数。
- ☑ 在结构体外分别实现结构体中定义的这些函数。
- ☑ 向内核中注册或删除驱动模块。

由此可见，实现字符设备驱动的首要任务是定义一个结构体。字符设备提供给应用程序流控制接口有 open、close、read、write 和 ioctl，添加一个字符设备驱动程序，实际上是给上述操作添加对应的代码，Linux 对这些操作统一做了抽象。

结构体 file_operations 定义格式如下所示。

```
static struct file_operations myDriver_fops = {
        owner: THIS_MODULE,
        write: myDriver_write,
        read:   myDriver_read,
        ioctl: myDriver_ioctl,
        open:   myDriver_open,
        release: myDriver_release,
};
```

在此结构体中规定了驱动程序向应用程序提供的操作接口，主要有实现以下几个功能的接口。

（1）实现 write 操作

实现 write 操作就是从应用程序接收数据送到硬件，例如下面的代码。

```
static ssize_t myDriver_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos){
	size_t fill_size = count;
	PRINTK("myDriver write called!\n");
	PRINTK("\tcount=%d, pos=%d\n", count, (int)*f_pos);
	if(*f_pos >= sizeof(myDriver_Buffer))
	{
		PRINTK("[myDriver write]Buffer Overlap\n");
		*f_pos = sizeof(myDriver_Buffer);
		return 0;
	}
	if((count + *f_pos) > sizeof(myDriver_Buffer))
	{
		PRINTK("count + f_pos > sizeof buffer\n");
		fill_size = sizeof(myDriver_Buffer) - *f_pos;
	}
	copy_from_user(&myDriver_Buffer[*f_pos], buf, fill_size);
	*f_pos += fill_size;
	return fill_size;
}
```

在上述代码中，函数 u_long copy_from_user(void *to, const void *from, u_long len)用于把用户态的数据复制到内核态，实现数据的传送。

（2）实现 read 操作

实现 read 操作即从硬件读取数据并交给应用程序。例如下面的代码。

```
static ssize_t myDriver_read(struct file *filp, char *buf, size_t count, loff_t *f_pos){
	size_t read_size = count;
	PRINTK("myDriver read called!\n");
	PRINTK("\tcount=%d, pos=%d\n", count, (int)*f_pos);
	if(*f_pos >= sizeof(myDriver_Buffer))
	{
		PRINTK("[myDriver read]Buffer Overlap\n");
		*f_pos = sizeof(myDriver_Buffer);
		return 0;
	}
	if((count + *f_pos) > sizeof(myDriver_Buffer))
	{
		PRINTK("count + f_pos > sizeof buffer\n");
		read_size = sizeof(myDriver_Buffer) - *f_pos;
	}
	copy_to_user(buf, &myDriver_Buffer[*f_pos], read_size);
	*f_pos += read_size;
	return read_size;
}
```

在上述代码中，函数 u_long copy_to_user(void * to, const void *from, u_long len)用于实现把内核态的数据复制到用户态下。

（3）实现 ioctl 操作

实现 ioctl 操作即为应用程序提供对硬件行为的控制，例如下面的代码。

```
static int myDriver_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg){
    PRINTK("myDriver ioctl called(%d)!\n", cmd);
    if(_IOC_TYPE(cmd) != TSTDRV_MAGIC)
    {
        return -ENOTTY;
    }
    if(_IOC_NR(cmd) >= TSTDRV_MAXNR)
    {
        return -ENOTTY;
    }
    switch(cmd)
    {
      case MYDRV_IOCTL0:
            PRINTK("IOCTRL 0 called(0x%lx)!\n", arg);
            break;
      case MYDRV_IOCTL1:
            PRINTK("IOCTRL 1 called(0x%lx)!\n", arg);
            break;
      case MYDRV_IOCTL2:
            PRINTK("IOCTRL 2 called(0x%lx)!\n", arg);
            break;
      case MYDRV_IOCTL3:
            PRINTK("IOCTRL 3 called(0x%lx)!\n", arg);
            break;
      }
      return 0;
}
```

（4）实现 open 操作

当应用程序打开设备时对设备进行初始化，使用 MOD_INC_USE_COUNT 增加驱动程序的使用次数，例如下面的代码。

```
static int myDriver_open(struct inode *inode, struct file *filp){
      MOD_INC_USE_COUNT;
      PRINTK("myDriver open called!\n");
      return 0;
}
```

（5）实现 release 操作

当应用程序关闭设备时处理设备的关闭操作，使用 MOD_DEC_USE_COUNT 来增加驱动程序的使用次数，例如下面的代码。

```
static int myDriver_release(struct inode *inode, struct file *filp){
      MOD_DEC_USE_COUNT;
      PRINTK("myDriver release called!\n");
      return 0;
}
```

（6）驱动程序初始化函数

Linux 在加载内核模块时会调用初始化函数，初始化驱动程序本身使用 register_chrdev 向内核注册驱动程序，该函数的第三个参数指向包含有驱动程序接口函数信息的 file_operations 结构体，例如下面的代码。

```
#ifdef CONFIG_DEVFS_FS
devfs_handle_t devfs_myDriver_dir;
devfs_handle_t devfs_myDriver_raw;
#endif
static int __init myModule_init(void)
{
	/* Module init code */
	PRINTK("myModule_init\n");
	/* Driver register */
	myDriver_Major = register_chrdev0,DRIVER_NAME,&myDriver_fops);
	if(myDriver_Major < 0)
	{
		PRINTK("register char device fail!\n");
		return myDriver_Major;
	}
	PRINTK("register myDriver OK! Major = %d\n", myDriver_Major);
#ifdef CONFIG_DEVFS_FS
	devfs_myDriver_dir = devfs_mk_dir(NULL, "myDriver", NULL);
	devfs_myDriver_raw = devfs_register(devfs_myDriver_dir, "raw0", DEVFS_FL_DEFAULT, myDriver_
Major, 0, S_IFCHR | S_IRUSR | S_IWUSR, &myDriver_fops, NULL);
	PRINTK("add dev file to devfs OK!\n");
#endif
	return 0;
}
```

在上述代码中，函数 module_init()用于向内核声明当前模块的初始化函数。

（7）驱动程序退出函数

Linux 在卸载内核模块时会调用退出函数释放驱动程序使用的资源，使用 unregister_chrdev 从内核中卸载驱动程序。将驱动程序模块注册到内核，内核需要知道模块的初始化函数和退出函数，才能将模块放入自己的管理队列中，例如下面的代码。

```
static void __exit myModule_exit(void){
	/* Module exit code */
	PRINTK("myModule_exit\n");
	/* Driver unregister */
	if(myDriver_Major > 0)
	{
	    #ifdef CONFIG_DEVFS_FS
	    devfs_unregister(devfs_myDriver_raw);
	    devfs_unregister(devfs_myDriver_dir);
	    #endif
	    unregister_chrdev(myDriver_Major, DRIVER_NAME);
	}
	return;
}
```

在上述代码中，函数 module_exit()用于向内核声明当前模块的退出函数。

经过上述分析，可以总结出开发字符设备驱动程序的基本步骤如下。

- ☑ 确定主设备号和次设备号。
- ☑ 实现字符驱动程序，先实现 file_operations 结构体，然后实现初始化函数并注册字符设备，接下来实现销毁函数并释放字符设备。

☑ 创建设备文件节点。

接下来给出一个通用的字符设备驱动程序，此程序由如下两个文件构成，其中文件 tst-driver.h 的实现代码如下所示。

```
#ifndef __TST_DRIVER_H__
#define __TST_DRIVER_H__
#define TSTDRV_MAGIC                    0xd0
#define GPIO_IN                         0
#define GPIO_OUT                        1//_IO(TSTDRV_MAGIC, 1)
#define GPIO_SET_BIT                    2//_IO(TSTDRV_MAGIC, 2)
#define GPIO_CLR_BIT                    3//_IO(TSTDRV_MAGIC, 3)
#define TSTDRV_MAXNR                    4
#endif    //#ifndef __TST_DRIVER_H__
```

另一个构成文件 tst-driver.c 的实现代码如下所示。

```
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
    #define MODULE
#endif
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>   /* printk() */
#include <linux/init.h>      /* __init __exit */
#include <linux/types.h>     /* size_t */
#include <linux/fs.h>        /* file_operation */
//#include <linux/errno.h> /* Error number */
//#include <linux/delay.h> /* udelay */
#include <asm/uaccess.h>     /* copy_to_user, copy_from_user */
#include <asm/hardware.h>
#include "tst-driver.h"
#define DRIVER_NAME "myDriver"
//#undef CONFIG_DEVFS_FS
#ifdef DEBUG
#define PRINTK(fmt, arg...) printk(KERN_NOTICE fmt, ##arg)
#else
#define PRINTK(fmt, arg...)
#endif
/*      KERN_EMERG 用于紧急事件，一般是系统崩溃前的提示信息
        KERN_ALERT 用于需要立即采取动作的场合
        KERN_CRIT 临界状态，通常设计验证的硬件或软件操作失败
        KERN_ERR 用于报告错误状态，设备驱动程序通常会用它报告来自硬件的问题
        KERN_WARNING 就可能出现的问题提出警告，这些问题通常不会对系统造成严重破坏
        KERN_NOTICE 有必要提示的正常情况。许多安全相关的情况用这个级别汇报
        KERN_INFO 提示性信息。有很多驱动程序在启动时用这个级别打印相关信息
        KERN_DEBUG 用于调试的信息
*/
static int myDriver_Major = 0; /* Driver Major Number */
/* Vitual Driver Buffer */
static unsigned char myDriver_Buffer[1024*1024];
/* Driver Operation Functions */
```

```
static int myDriver_open(struct inode *inode, struct file *filp)
{
// int Minor = MINOR(inode->i_rdev);
// filp->private_data = 0;
        MOD_INC_USE_COUNT;
        PRINTK("myDriver open called!\n");
        return 0;
}
static int myDriver_release(struct inode *inode, struct file *filp)
{
// int Minor = MINOR(inode->i_rdev);
        MOD_DEC_USE_COUNT;
        PRINTK("myDriver release called!\n");
        return 0;
}
static ssize_t myDriver_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
        size_t read_size = count;
        PRINTK("myDriver read called!\n");
        PRINTK("\tcount=%d, pos=%d\n", count, (int)*f_pos);
        if(*f_pos >= sizeof(myDriver_Buffer))
        {
                PRINTK("[myDriver read]Buffer Overlap\n");
                *f_pos = sizeof(myDriver_Buffer);
                return 0;
        }
        if((count + *f_pos) > sizeof(myDriver_Buffer))
        {
                PRINTK("count + f_pos > sizeof buffer\n");
                read_size = sizeof(myDriver_Buffer) - *f_pos;
        }
        copy_to_user(buf, &myDriver_Buffer[*f_pos], read_size);
        *f_pos += read_size;
        return read_size;
}
static ssize_t myDriver_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos)
{
        size_t fill_size = count;
        PRINTK("myDriver write called!\n");
        PRINTK("\tcount=%d, pos=%d\n", count, (int)*f_pos);
        if(*f_pos >= sizeof(myDriver_Buffer))
        {
                PRINTK("[myDriver write]Buffer Overlap\n");
                *f_pos = sizeof(myDriver_Buffer);
                return 0;
        }
        if((count + *f_pos) > sizeof(myDriver_Buffer))
        {
                PRINTK("count + f_pos > sizeof buffer\n");
                fill_size = sizeof(myDriver_Buffer) - *f_pos;
        }
```

```
            copy_from_user(&myDriver_Buffer[*f_pos], buf, fill_size);
            *f_pos += fill_size;
            return fill_size;
}
static int myDriver_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
       PRINTK("myDriver ioctl called(%d)!\n", cmd);
       if(_IOC_TYPE(cmd) != TSTDRV_MAGIC)
       {
              return -ENOTTY;
       }
       if(_IOC_NR(cmd) >= TSTDRV_MAXNR)
       {
              return -ENOTTY;
       }
       switch(cmd)
            {
              case MYDRV_IOCTL0:
                      PRINTK("IOCTRL 0 called(0x%lx)!\n", arg);
                      break;
              case MYDRV_IOCTL1:
                      PRINTK("IOCTRL 1 called(0x%lx)!\n", arg);
                      break;
              case MYDRV_IOCTL2:
                      PRINTK("IOCTRL 2 called(0x%lx)!\n", arg);
                      break;
              case MYDRV_IOCTL3:
                      PRINTK("IOCTRL 3 called(0x%lx)!\n", arg);
                      break;
            }
            return 0;
}
/* 驱动操作结构 */
static struct file_operations myDriver_fops = {
            owner: THIS_MODULE,
            write: myDriver_write,
            read: myDriver_read,
            ioctl: myDriver_ioctl,
            open: myDriver_open,
            release: myDriver_release,
};
/* 分别定义模块初始化和退出代码 */
#ifdef CONFIG_DEVFS_FS
devfs_handle_t devfs_myDriver_dir;
devfs_handle_t devfs_myDriver_raw;
#endif
static int __init myModule_init(void)
{
            /* 模块初始化 */
            PRINTK("myModule_init\n");
            /* Driver register */
```

```
        myDriver_Major = register_chrdev(0, DRIVER_NAME, &myDriver_fops);
        if(myDriver_Major < 0)
        {
                PRINTK("register char device fail!\n");
                return myDriver_Major;
        }
        PRINTK("register myDriver OK! Major = %d\n", myDriver_Major);
#ifdef CONFIG_DEVFS_FS
        devfs_myDriver_dir = devfs_mk_dir(NULL, "myDriver", NULL);
        devfs_myDriver_raw = devfs_register(devfs_myDriver_dir, "raw0", DEVFS_FL_DEFAULT, myDriver_
Major, 0, S_IFCHR | S_IRUSR | S_IWUSR, &myDriver_fops, NULL);
        PRINTK("add dev file to devfs OK!\n");
#endif
        return 0;
}
static void __exit myModule_exit(void)
{
        /* 退出模块 */
        PRINTK("myModule_exit\n");
        /* 取消驱动 */
        if(myDriver_Major > 0)
        {
#ifdef CONFIG_DEVFS_FS
                devfs_unregister(devfs_myDriver_raw);
                devfs_unregister(devfs_myDriver_dir);
#endif
                unregister_chrdev(myDriver_Major, DRIVER_NAME);
        }
        return;
}
MODULE_AUTHOR("SXZ");
MODULE_LICENSE("Dual BSD/GPL");
module_init(myModule_init);
module_exit(myModule_exit);
```

2.6.2　块设备驱动

块设备 I/O 与字符设备操作的主要区别如下。

☑ 块设备只能以块为单位接收输入返回输出，而字符设备则以 byte 为单位。大多数设备是字符设备，它们不需要缓冲并且不以固定块大小进行操作。

☑ 块设备对于 I/O 请求有对应的缓冲区，所以它们可以选择以什么顺序进行响应。字符设备无须缓冲且被直接读写。

☑ 字符设备只能被顺序读写，块设备可以随机访问。

（1）结构体 block_device_operations

在文件 include/linux/fs.h 中定义了结构体 block_device_operations，此结构体描述了对块设备的操作集合，具体代码如下所示。

```
struct block_device_operations {
        int (*open) (struct inode *, struct file *);                /*打开*/
```

```
	int (*release) (struct inode *, struct file *);		/*释放*/
	int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
	long (*unlocked_ioctl) (struct file *, unsigned, unsigned long);
	long (*compat_ioctl) (struct file *, unsigned, unsigned long);
	int (*direct_access) (struct block_device *, sector_t, unsigned long *);
	int (*media_changed) (struct gendisk *);		/*介质被改变*/
	int (*revalidate_disk) (struct gendisk *);		/*使介质改变*/
	int (*getgeo)(struct block_device *, struct hd_geometry *);	/*填充驱动器信息*/
	struct module *owner;				/*模块拥有者，一般初始化为 THIS_MODULE*/
};
```

（2）结构体 gendisk

结构体 gendisk 描述一个独立的磁盘设备或分区，具体代码如下所示。

```
struct gendisk{
	/*前 3 个元素共同表征了一个磁盘的主、次设备号，同一个磁盘的各个分区共享一个主设备号*/
	int major;					/*主设备号*/
	int first_minor;				/*第一个次设备号*/
	int minors;					/*最大的次设备数，如果不能分区，则为 1*/
	char disk_name[32];
	struct hd_struct** part;			/*磁盘上的分区信息*/
	struct block_device_operations* fops;		/*块设备操作，block_device_operations*/
	struct request_queue* queue;			/*请求队列，用于管理该设备 I/O 请求队列的指针*/
	void* private_data;				/*私有数据*/
	sector_t capacity;				/*扇区数，512 字节为 1 个扇区，描述设备容量*/
	//...
};
```

（3）结构体 request 和 bio

☑ 请求 request：结构体 request 和 request_queue 在 Linux 块设备驱动中，使用结构体 request 表示等待进行的 I/O 请求，用 request_queue 表示一个块 I/O 请求队列。定义这两个结构体的代码如下所示。

```
struct request{
	struct list_head queuelist;
	unsigned long flags;
	sector_t sector;				/*要传输的下一个扇区*/
	unsigned long nr_sectors;			/*要传送的扇区数目*/
	unsigned int current_nr_sector;			/*当前要传送的扇区*/
	sector_t hard_sector;				/*要完成的下一个扇区*/
	unsigned long hard_nr_sectors;			/*要被完成的扇区数目*/
	unsigned int hard_cur_sectors;			/*当前要被完成的扇区数目*/
	struct bio* bio;				/*请求的 bio 结构体的链表*/
	struct bio* biotail;				/*请求的 bio 结构体的链表尾*/
	/*请求在物理内存中占据的不连续的段的数目* /
	unsigned short nr_phys_segments;
	unsigned short nr_hw_segments;
	int tag;
	char* buffer;					/*传送的缓冲区，内核的虚拟地址*/
	int ref_count;					/*引用计数*/
	...
    };
```

☑ 请求队列 request_queue：请求队列跟踪等候的块 I/O 请求，它存储用于描述这个设备能够支持的请求的类型信息。请求队列还要实现一个插入接口，这个接口允许使用多个 I/O 调度器，I/O 调度

器以最优性能的方式向驱动提交 I/O 请求。大部分 I/O 调度器是积累批量的 I/O 请求，并将其排列为递增/递减的块索引顺序后提交给驱动。另外，I/O 调度器还负责合并邻近的请求，当一个新的 I/O 请求被提交给调度器后，它会在队列中搜寻包含邻近的扇区的请求。如果找到一个并且此请求合理，则调度器会将这两个请求合并。

定义结构体 request_queue 的代码如下所示。

```
struct request_queue{
        …
        /*自旋锁，保护队列结构体*/
        spinlock_t __queue_lock;
        spinlock_t* queue_lock;
        struct kobject kobj;                        /*队列 kobject*/
        /*队列设置*/
        unsigned long nr_requests;                  /*最大的请求数量*/
        unsigned int    nr_congestion_on;
        unsigned int    nr_congestion_off;
        unsigned int    nr_batching;
        unsigned short max_sectors;                 /*最大扇区数*/
        unsigned short max_hw_sectors;
        unsigned short max_phys_sectors;            /*最大的段数*/
        unsigned short max_hw_segments;
        unsigned short hardsect_size;               /*硬件扇区尺寸*/
        unsigned int max_segment_size;              /*最大的段尺寸*/
        unsigned long seg_boundary_mask;            /*段边界掩码*/
        unsigned int dma_alignment;                 /*DMA 传送内存对齐限制*/
        struct blk_queue_tag* queue_tags;
        atomic_t refcnt;                            /*引用计数*/
        unsigned int in_flight;
        unsigned int sg_timeout;
        unsigned int sg_reserved_size;
        int node;
        struct list_head drain_list;
        struct request* flush_rq;
        unsigned char ordered;
};
```

注意： 在块设备模块中，还可以使用函数实现块设备驱动的模块卸载、加载、打开与释放操作，相关知识请参阅相关资料。

Android 的块设备驱动在目录/dev/block 中，其主要内容如下所示。

```
brw------- root      root      179,   2 2012-02-29 23:33 mmcblk0p2
brw------- root      root      179,   1 2012-02-29 23:33 mmcblk0p1
brw------- root      root      179,   0 2012-02-29 23:33 mmcblk0
brw------- root      root       31,   6 2012-02-29 23:33 mtdblock6
brw------- root      root       31,   5 2012-02-29 23:33 mtdblock5
brw------- root      root       31,   4 2012-02-29 23:33 mtdblock4
brw------- root      root       31,   3 2012-02-29 23:33 mtdblock3
brw------- root      root       31,   2 2012-02-29 23:33 mtdblock2
brw------- root      root       31,   1 2012-02-29 23:33 mtdblock1
brw------- root      root       31,   0 2012-02-29 23:33 mtdblock0
brw------- root      root        7,   7 2012-02-29 23:33 loop7
```

```
brw------- root     root      7,   6 2012-02-29 23:33 loop6
brw------- root     root      7,   5 2012-02-29 23:33 loop5
brw------- root     root      7,   4 2012-02-29 23:33 loop4
brw------- root     root      7,   3 2012-02-29 23:33 loop3
brw------- root     root      7,   2 2012-02-29 23:33 loop2
brw------- root     root      7,   1 2012-02-29 23:33 loop1
brw------- root     root      7,   0 2012-02-29 23:33 loop0
brw------- root     root      1,   0 2012-02-29 23:33 ram0
brw------- root     root      1,   0 2012-02-29 23:33 ram1
```

在上述内容中，主设备号为 1 的是各个内存块设备，主设备号为 7 的是各个回环块设备，主设备号为 31 的是 mtd 设备中的块设备，mmcblk0 表示 SD 卡的块设备。

在 Android 系统中，可以使用 mount 命令来查看系统中被挂起的文件系统。使用 mount 命令的格式如下所示。

```
mount [-t vfstype] [-o options] device dir
```

上述命令中的主要参数的具体说明如下。

（1）-t vfstype：用于指定文件系统的类型，通常不必指定。mount 会自动选择正确的类型。常用类型有如下 6 类。

- ☑ 光盘或光盘镜像：iso9660。
- ☑ DOS fat16 文件系统：msdos。
- ☑ Windows 9x fat32 文件系统：vfat。
- ☑ Windows NT ntfs 文件系统：ntfs。
- ☑ Mount Windows 文件网络共享：smbfs。
- ☑ UNIX(LINUX) 文件网络共享：nfs。

（2）-o options：主要用来描述设备或档案的挂接方式，其中常用的参数有如下 4 类。

- ☑ loop：用来把一个文件当成硬盘分区挂接上系统。
- ☑ ro：采用只读方式挂接设备。
- ☑ rw：采用读写方式挂接设备。
- ☑ iocharset：指定访问文件系统所用字符集。

（3）device：要挂接（mount）的设备。

（4）dir：设备在系统上的挂接点（mount point）。

另外，在 Android 系统中可以使用 df 命令来查看系统中各个盘的使用情况。使用 df 命令的格式如下所示。

```
df [options]
```

参数 options 常用取值的具体说明如下。

- ☑ －s：对每个 Names 参数只给出占用的数据块总数。
- ☑ －a：递归地显示指定目录中各文件及子目录中各文件占用的数据块数。若既不指定－s，也不指定－a，则只显示 Names 中的每一个目录及其中的各子目录所占的磁盘块数。
- ☑ －k：以 1024 字节为单位列出磁盘空间使用情况。
- ☑ －x：跳过在不同文件系统上的目录不予统计。
- ☑ －l：计算所有的文件大小，对硬链接文件则计算多次。
- ☑ －i：显示 inode 信息而非块使用量。
- ☑ －h：以容易理解的格式印出文件系统大小，例如 136KB、254MB、21GB。
- ☑ －P：使用 POSIX 输出格式。
- ☑ －T：显示文件系统类型。

2.6.3　网络设备驱动

Linux 网络设备驱动程序由 4 部分组成，分别是网络设备媒介层、网络设备驱动层、网络设备接口层以及网络协议接口层。网络设备媒介层包括各种物理网络设备和传输媒介。对于网络设备接口层，Linux 系统用 Net_device 结构表示网络设备接口。Net_device 结构保存所有与硬件有关的接口信息，各协议软件主要通过 Net_device 结构来完成与硬件的交互。网络设备驱动层主要包括网络设备的初始化、数据包的发送和接收。网络协议接口层提供网络接口驱动程序的抽象接口。

在 Linux 网络驱动程序中，常用方法如下。

（1）初始化（initialize）：检测设备；配置和初始化硬件；初始化 net_device 结构；注册设备。

（2）打开（open）：这个方法在网络设备被激活的时候被调用。进行资源的申请和硬件的激活等。open 方法另一个作用是如果驱动程序作为一个模块被装入，则要防止模块卸载时设备处于打开状态。在 open 方法中要调用 MOD_INC_USE_COUNT 宏。

（3）关闭（close）：释放某些系统资源。如果是作为模块装入的驱动程序，close 中应该调用 MOD_DEC_USE_COUNT，减少设备被引用的次数，以使驱动程序可以被卸载。

（4）发送（hard_start_xmit）：网络设备驱动程序发送数据时，系统调用 dev_queue_xmit 函数，发送的数据放在一个 sk_buff 结构中。一般的驱动程序将数据传输到硬件发出去, 特殊的设备如 loopback 把数据组成一个接收数据再回送给系统，或如 dummy 设备直接丢弃数据。如果发送成功，则在 hard_start_xmit 方法中释放 sk_buff，返回 0，否则返回 1。

（5）接收（reception）：驱动程序并不存在一个接收方法。有数据收到应该是驱动程序来通知系统的。一般设备收到数据后都会产生一个中断，在中断处理程序中驱动程序申请一块 sk_buff，从硬件读出数据放置到申请好的缓冲区中。接下来填充 sk_buff 中的一些信息。最后调用 netif_rx()把数据传送给上层协议层处理。

在 Android 系统中，可以使用 ifconfig 命令来查询系统中的网络设备，另外使用此命令也可以获取 WiFi 网络和电话网络的信息。

2.7　Android 系统移植基础

本书讲解的是 Android 驱动开发，由图 2-1 可知，驱动开发是最底层的应用，属于 Linux 内核层的工作。因为驱动是系统和硬件之间的载体，涉及不同硬件的应用问题，所以需要做系统移植的工作。本节将简要介绍 Android 系统移植方面的有关问题。

2.7.1　移植的任务

Android 移植开发的最终目的是为了开发手机产品，从开发者的角度来看，这种类型的开发以具有硬件系统为前提，在硬件系统的基础上构建 Android 软件系统。这种类型的开发工作在 Android 系统的底层。在软件系统方面，主要的工作集中在以下两个方面。

（1）Linux 中的相关设备驱动程序

驱动程序是硬件和上层软件的接口，在 Android 手机系统中，需要基本的屏幕、触摸屏、键盘等驱动程序，以及音频、摄像头、电话的 Modem、WiFi、蓝牙等多种设备驱动程序。

（2）Android 本地框架中的硬件抽象层

在 Android 中，硬件抽象层工作在用户空间，介于驱动程序和 Android 系统之间。Android 系统对硬件抽象层通常都有标准的接口定义，在开发过程中，实现这些接口也就给 Android 系统提供了硬件抽象层。

上述两个部分综合起来相互结合，共同完成了 Android 系统的软件移植。移植成功与否取决于驱动程序的品质和对 Android 硬件抽象层接口的理解程度。Android 移植开发的工作由核心库、Dalvik 虚拟机/ART、硬件抽象层、Linux 内核层和硬件系统协同完成的，具体结构如图 2-10 所示。

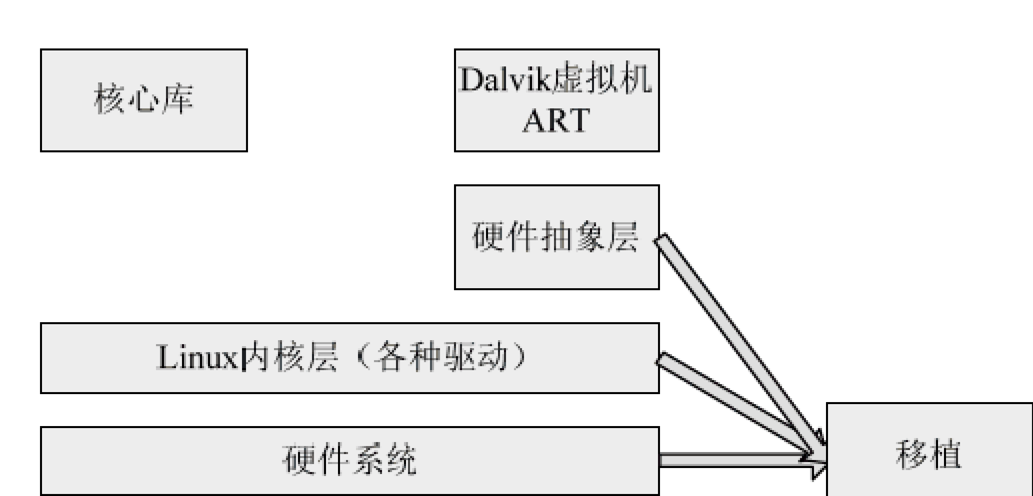


图 2-10　Android 移植结构

2.7.2　需要移植的内容

在 Android 系统中，在移植过程中主要移植驱动方面的内容。Android 的移植主要可以分为如下几个类型。




（1）基本图形用户界面（GUI）部分：包括显示部分、用户输入部分和硬件相关的加速部分，还包括媒体编解码和 OpenGL 等。

（2）音视频输入/输出部分：包括音频、视频输出和摄像头等。

（3）连接部分：包括无线局域网、蓝牙、GPS 等。

（4）电话部分：包括通话、GSM 等。

（5）附属部件：包括传感器、背光、振动器等。

具体来说主要移植下面的内容。

（1）Display 显示部分：包括 FrameBuffer 驱动和 Gralloc 模块。

（2）Input 用户输入部分：包括 Event 驱动和 EventHub。

（3）Codec 多媒体编解码：包括硬件 Codec 驱动和 Codec 插件，例如 OpenMax。

（4）3D Accelerator（3D 加速器）部分：包括硬件 OpenGL 驱动和 OpenGL 插件。

（5）Audio 音频部分：包括 Audio 驱动和 Audio 硬件抽象层。

（6）Video Out 视频输出部分：包括视频显示驱动和 Overlay 硬件抽象层。

（7）Camera 摄像头部分：包括 Camera 驱动（通常是 v4l2）和 Camera 硬件抽象层。

（8）Phone 电话部分：包括 Modem 驱动程序和 RIL 库。

（9）GPS 全球定位系统部分：包括 GPS 驱动（例如串口）和 GPS 硬件抽象层。

（10）WiFi 无线局域网部分：包括 Wlan 驱动协议和 WiFi 的适配层。

（11）BlueTooth 蓝牙部分：包括 BT 驱动协议和 BT 的适配层。

（12）Sensor 传感器部分：包括 Sensor 驱动和 Sensor 硬件抽象层。

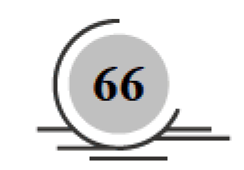
（13）Vibrator 振动器部分：包括 Vibrator 驱动和 Vibrator 硬件抽象层。

（14）Light 背光部分：包括 Light 驱动和 Light 硬件抽象层。

（15）Alarm 警告器部分：包括 Alarm 驱动、RTC 系统和用户空间调用。

（16）Battery 电池部分：包括电池部分驱动和电池的硬件抽象层。

注意： Android 系统有很多组件，但并不是每一个组件都需要移植，例如那些纯软的组件就不需要移植。另外，浏览器引擎虽然需要下层的网络支持，但是实际上并不需要直接为其移植网络接口，而是通过无线局域网或者电话系统数据连接来完成标准的网络接口。



2.7.3　驱动开发需要做的工作

在移植 Android 系统驱动的过程中，我们的任务就是为某一个将要在 Android 系统上使用的硬件开发一个驱动程序。因为 Android 是基于 Linux 的，所以开发 Android 驱动其实就是开发 Linux 驱动。对于大部分子系统来说，硬件抽象层和驱动程序都需要根据实际系统的情况来实现，例如传感器部分、音频部分、视频部分、摄像头部分和电话部分。另外也有一些子系统的硬件抽象层是标准的，只需要实现 Linux 内核中的驱动程序即可，例如输入部分、振动器部分、无线局域网部分和蓝牙部分等。对于有标准的硬件抽象层的系统，有时也需要做一些配置工作。

随着 Android 系统的更新和发展，它已经不仅仅是一个移动设备的平台，也可以用于消费类电子和智能家电，例如 3.0 以后的版本主要是针对平板电脑的，另外，电子书、数字电视、机顶盒、固定电话等也逐渐开始使用 Android 系统。在这些平台上，通常需要实现比移动设备更少的部件。一般来说，包括显示和用户输入的基本用户界面部分是需要移植的，其他部分是可选的。例如电话系统、振动器、背光、传感器等一般不需要在非移动设备系统来实现，一些固定位置设备通常不需要实现 GPS 系统。

2.8　内核空间和用户空间之间的接口

在 Android 驱动开发应用中，我们编写的驱动程序是供系统硬件使用的，也就是说，驱动程序是介于系统和硬件之间的桥梁。在 Linux 环境下开发这些中间桥梁的驱动程序时，需要用到内核空间和用户空间之间的接口，本节将详细讲解内核空间和用户空间之间的接口的基本知识。

2.8.1　内核空间和用户空间的相互作用

在现实 Android 开发过程中，越来越多的应用程序需要编写内核级和用户级的程序来一起完成某个任务，这通常采用以下流程来实现。

（1）编写内核服务程序利用内核空间提供的权限和服务来接收、处理和缓存数据。

（2）编写用户程序和之前完成的内核服务程序交互，具体来说，可以利用用户程序来配置内核服务程序的参数，提取内核服务程序提供的数据。另外，也可以向内核服务程序输入待处理数据。

在现实 Android 开发过程中，需要内核空间和用户空间联合完成的典型应用有：Netfilter（内核服务程序：防火墙）VS Iptable（用户级程序：规则设置程序）；IPSEC（内核服务程序：VPN 协议部分）VS IKE（用户级程序：vpn 密钥协商处理）；当然还包括大量的设备驱动程序及相应的应用软件。

2.8.2　实现系统和硬件之间的交互

在 Android 底层开发应用中，实现硬件和系统的交互是我们的主要任务之一。在 Linux 平台下，有如下 5 种实现硬件和系统的交互功能的方式。

（1）编写自己的系统调用

系统调用是用户级程序访问内核最基本的方法。目前 Linux 大致提供了 200 多个标准的系统调用（具体请参考内核代码树中的 include/asm-i386/unistd.h 和 arch/i386/kernel/entry.S 文件），并且允许我们添加自己的系统调用来实现和内核的信息交换。假如我们想建立一个系统调用日志系统，将所有的系统调用动作记

录下来，以便进行入侵检测，此时可以编写一个内核服务程序，该程序负责收集所有的系统调用请求，并将这些调用信息记录到在内核中自建的缓冲中。我们无法在内核中实现复杂的入侵检测程序，因此必须将该缓冲中的记录提取到用户空间。最直接的方法是自己编写一个新系统调用实现这种提取缓冲数据的功能。当内核服务程序和新系统调用都实现后，就可以在用户空间中编写用户程序进行入侵检测任务了，入侵检测程序可以定时、轮询或在需要时调用新系统从内核提取数据，然后进行入侵检测。

（2）编写驱动程序

Linux/UNIX 的一个特点就是把所有的东西都看作文件（every thing is a file）。系统定义了简洁完善的驱动程序界面，客户程序可以用统一的方法通过这个界面和内核驱动程序交互。而大部分系统的使用者和开发者已经非常熟悉这种界面以及相应的开发流程了。

驱动程序运行于内核空间，用户空间的应用程序通过文件系统中/dev/目录下的一个文件来和它交互，这就是传统的文件操作流程：open()→read()→write()→ioctl()→close()。

注意： 并不是所有的内核驱动程序都是这个界面，网络驱动程序和各种协议栈的使用就不一致，如套接口编程虽然也有 open()close()等概念，但它的内核实现以及外部使用方式都和普通驱动程序有很大差异。

这里先不谈设备驱动程序在内核中要做的中断响应、设备管理、数据处理等工作，在此先把注意力集中在它与用户级程序交互这一部分。操作系统为此定义了一种统一的交互界面，就是前面所说的 open()、read()、write()、ioctl()和 close()等。每个驱动程序按照自己的需要做独立实现，把自己提供的功能和服务隐藏在这个统一界面下。客户级程序选择需要的驱动程序或服务（其实就是选择/dev/目录下的文件），按照上述界面和文件操作流程，就可以和内核中的驱动交互了。其实用面向对象的概念会更容易解释，系统定义了一个抽象的界面（abstract interface），每个具体的驱动程序都是这个界面的实现（implementation）。

由此可见，驱动程序也是用户空间和内核信息交互的重要方式之一。从本质上来说，ioctl、read、和 write 也是通过系统调用去完成的，只是这些调用已被内核进行了标准封装和统一定义。因此用户不必像添加新系统调用那样必须修改内核代码，重新编译新内核，使用虚拟设备只需要通过模块方法将新的虚拟设备安装到内核中（insmod 上）就能方便使用。

可以将 Linux 中的设备大致分为如下 3 类。

- ☑ 字符设备：包括那些必须以顺序方式，像字节流一样被访问的设备。
- ☑ 块设备：是指那些可以用随机方式，以整块数据为单位来访问的设备，如硬盘等。
- ☑ 网络接口：指通常网卡和协议栈等复杂的网络输入/输出服务。

如果将我们的系统调用日志系统用字符型驱动程序的方式实现，整个过程就非常简单了。我们可以将内核中收集和记录信息的那一部分编写成一个字符设备驱动程序。虽然没有实际对应的物理设备，但是 Linux 的设备驱动程序本来就是一个软件抽象，它可以结合硬件提供服务，也完全可以作为纯软件提供服务。在驱动程序中，可以使用 open()来启动服务，用 read()返回处理好的记录，用 ioctl()设置记录格式等，用 close()停止服务，而 write()没有用到，那么我们可以不去实现它。然后在/dev/目录下建立一个设备文件，这个文件和新加入内核中的日志驱动系统程序相对应。

（3）使用 proc 文件系统

proc 是 Linux 提供的一种特殊的文件系统，使用它的目的就是提供一种便捷的用户和内核间的交互方式。proc 以文件系统作为使用界面，使应用程序可以以文件操作的方式安全、方便地获取系统当前运行的状态和其他一些内核数据信息。

proc 文件系统多用于监视、管理和调试系统，平常使用的 ps 和 top 等管理工具就是利用 proc 来读取内核信息的。除了读取内核信息外，proc 文件系统还提供了写入功能。所以我们也可以利用它来向内核输入信息。例如通过修改 proc 文件系统下的系统参数配置文件/proc/sys 后可以直接在运行时动态更改内核参数。

除了系统已经提供的文件条目，通过 proc 为我们留的接口可以允许在内核中创建新的条目从而与用户程序共享信息数据。例如可以为系统调用日志程序（无论是作为驱动程序还是作为单纯的内核模块）在 proc 文件系统中创建新的文件条目，在此条目中显示系统调用的使用次数及每个单独系统调用的使用频率等，也可以增加另外的条目用于设置日志记录规则。

（4）使用虚拟文件系统（VFS）

很多内核开发者认为利用 ioctl()系统调用往往会使系统调用意义不明确而且难以控制。而将信息放入 proc 文件系统中会使信息组织混乱，所以不赞成过多使用此系统。建议是实现一种孤立的虚拟文件系统来代替 ioctl()和 proc，这是因为文件系统接口清楚，而且便于用户空间访问，同时利用虚拟文件系统使得利用脚本执行系统管理任务更加方便、有效。

下面举例来说明如何通过虚拟文件系统修改内核信息。假设我们可以实现一个名为 sagafs 的虚拟文件系统，其中文件 log 对应内核存储的系统调用日志。此时就可以通过文件访问的普遍方法获得日志信息，命令如下所示。

```
# cat /sagafs/log
```

使用虚拟文件系统可以更加方便、清晰地实现信息交互。但是很多程序员认为 VFS 的 API 接口十分复杂，其实读者们无须担心，因为从 Linux 2.5 内核开始就提供了一种叫做 libfs 的例程序，帮助不熟悉文件系统的用户封装了实现 VFS 的通用操作。

（5）使用内存映像

Linux 通过内存映像机制来提供用户程序对内存直接访问的能力。内存映像的意思是把内核中特定部分的内存空间映射到用户级程序的内存空间去。也就是说，用户空间和内核空间共享一块相同的内存。这样做有如下影响。

内核在这块地址内存储变更的任何数据，用户可以立即发现和使用，根本无须数据复制。在使用系统调用交互信息时，在整个操作过程中必须有一步数据复制的工作，或者是把内核数据复制到用户缓冲区，或只是把用户数据复制到内核缓冲区。这样对许多数据传输量大、时间要求高的应用来说很不科学，因为许多应用根本就无法忍受数据复制所耗费的时间和资源。

2.8.3　从内核到用户空间传输数据

Relay 是一种从 Linux 内核到用户空间的高效数据传输技术。通过用户定义的 relay 通道，内核空间的程序能够高效、可靠、便捷地将数据传输到用户空间。Relay 特别适用于内核空间有大量数据需要传输到用户空间的情形，目前已经广泛应用在内核调试工具如 SystemTap 中。

1．Relay 发展

Relay 的前身是 RelayFS，即作为 Linux 的一个新型文件系统。2003 年 3 月，RelayFS 的第一个版本的代码被开发出来，在 7 月 14 日，第一个针对 2.6 内核的版本也开始提供下载。经过广泛的试用和改进，直到 2005 年 9 月，RelayFS 才被加入 mainline 内核（2.6.14）。同时，RelayFS 也被移植到 2.4 内核中。在 2006 年 2 月，从 2.6.17 开始，RelayFS 不再作为单独的文件系统存在，而是成为内核的一部分。它的源码也从 fs/目录下转移到 kernel/relay.c 中，名称中也从 RelayFS 改成了 Relay。

RelayFS 目前已经被越来越多的内核工具使用，包括内核调试工具 SystemTap、LTT，以及一些特殊的文件系统，例如 DebugFS。

2．Relay 的原理

Relay 提供了一种机制，使得内核空间的程序能够通过用户定义的 relay 通道（channel）将大量数据高

效地传输到用户空间。一个 relay 通道由一组和 CPU 一一对应的内核缓冲区组成。这些缓冲区又被称为 relay 缓冲区（buffer），其中的每一个在用户空间都用一个常规文件来表示，被叫做 relay 文件（file）。内核空间的用户可以利用 relay 提供的 API 接口来写入数据，这些数据会被自动写入当前的 CPU id 对应的那个 relay 缓冲区；同时，这些缓冲区从用户空间看来是一组普通文件，可以直接使用 read()进行读取，也可以使用 mmap()进行映射。Relay 并不关心数据的格式和内容，这些完全依赖于使用 relay 的用户程序。Relay 的目的是提供一个足够简单的接口，从而使得基本操作尽可能的高效。

Relay 对数据实现了读和写的分离，使得突发性大量数据写入时不受限于用户空间相对较慢的读取速度，从而大大提高了效率。Relay 作为写入和读取的桥梁，也就是将内核用户写入的数据缓存并转发给用户空间的程序，这种转发机制也正是 Relay 这个名称的由来。

3．Relay 的 API

Relay 中提供了许多 API 来支持用户程序完整地使用 relay。这些 API 主要分为两大类，分别是按照面向用户空间和面向内核空间，具体说明如下。

（1）面向用户空间的 API

此类 API 编程接口向用户空间程序提供了访问 relay 通道缓冲区数据的基本操作的入口，主要包括如下方法。

☑ open()：允许用户打开一个已经存在的通道缓冲区。
☑ mmap()：使通道缓冲区被映射到位于用户空间的调用者的地址空间。要特别注意的是，我们不能仅对局部区域进行映射。也就是说，必须映射整个缓冲区文件，其大小是 CPU 的个数和单个 CPU 缓冲区大小的乘积。
☑ read()：读取通道缓冲区的内容。这些数据一旦被读出，就意味着它们被用户空间的程序消费掉了，也就不能被之后的读操作看到。
☑ sendfile()：将数据从通道缓冲区传输到一个输出文件描述符，其中可能的填充字符会被自动去掉，不会被用户看到。
☑ poll()：支持 POLLIN/POLLRDNORM/POLLERR 信号，每次子缓冲区的边界被越过时，等待着的用户空间程序会得到通知。
☑ close()：将通道缓冲区的引用数减 1。当引用数减为 0 时，表明没有进程或者内核用户需要打开它，从而这个通道缓冲区被释放。

（2）面向内核空间的 API

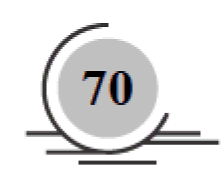
此类 API 接口向位于内核空间的用户提供了管理 relay 通道、数据写入等功能，其中最常用的如下。

☑ relay_open()：创建一个 relay 通道，包括创建每个 CPU 对应的 relay 缓冲区。
☑ relay_close()：关闭一个 relay 通道，包括释放所有的 relay 缓冲区，在此之前会调用 relay_switch()来处理这些 relay 缓冲区以保证已读取但是未满的数据不会丢失。
☑ relay_write()：将数据写入当前 CPU 对应的 relay 缓冲区内。由于它使用了 local_irqsave()保护，因此也可以在中断上下文中使用。
☑ relay_reserve()：在 relay 通道中保留一块连续的区域来留给未来的写入操作，这通常用于那些希望直接写入到 relay 缓冲区的用户。考虑到性能或者其他因素，这些用户不希望先把数据写到一个临时缓冲区中，然后再通过 relay_write()进行写入。

4．使用 Relay

下面将通过一个最简单的例子来介绍使用 Relay 的方法，本实例由如下两部分组成。

☑ 位于内核空间将数据写入 relay 文件的程序，使用时需要作为一个内核模块被加载。



☑　位于用户空间从 relay 文件中读取数据的程序，使用时作为普通用户状态的程序运行。

（1）实现内核空间

内核空间程序的主要操作如下。

☑　当加载模块时，打开一个 relay 通道，并且向打开的 relay 通道中写入消息。

☑　当卸载模块时，关闭 relay 通道。

实现文件 hello-mod.c 的具体实现代码如下所示。

```
#include <linux/module.h>
#include <linux/relayfs_fs.h>
static struct rchan *hello_rchan;
int init_module(void)
{
	const char *msg="Hello world\n";
	hello_rchan = relay_open("cpu", NULL, 8192, 2, NULL);
	if(!hello_rchan){
		printk("relay_open() failed.\n");
		return -ENOMEM;
	}
	relay_write(hello_rchan, msg, strlen(msg));
	return 0;
}
void cleanup_module(void)
{
	if(hello_rchan) {
		relay_close(hello_rchan);
		hello_rchan = NULL;
	}
	return;
}
MODULE_LICENSE ("GPL");
MODULE_DESCRIPTION ("Simple example of Relay");
```

（2）实现用户空间

用户空间的函数主要操作过程如下。

如果 relayfs 文件系统还没有被 umount（是一个命令），则将其 umount 到/mnt/relay 目录下。首先遍历每一个 CPU 对应的缓冲文件，然后打开文件，接着读取所有文件内容，然后关闭文件，最后 umount 掉 relay 文件系统。

实现文件 audience.c 的具体实现代码如下所示。

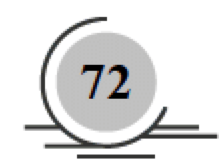
```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mount.h>
#include <fcntl.h>
#include <sched.h>
#include <errno.h>
#include <stdio.h>
#define MAX_BUFLEN 256
const char filename_base[]="/mnt/relay/cpu";
// implement your own get_cputotal() before compilation
static int get_cputotal(void);
int main(void)
```

```
{
	char filename[128]={0};
	char buf[MAX_BUFLEN];
	int fd, c, i, bytesread, cputotal = 0;
	if(mount("relayfs", "/mnt/relay", "relayfs", 0, NULL)
				&& (errno != EBUSY)) {
		printf("mount() failed: %s\n", strerror(errno));
		return 1;
	}
	cputotal = get_cputotal();
	if(cputotal <= 0) {
		printf("invalid cputotal value: %d\n", cputotal);
		return 1;
	}
	for(i=0; i<cputotal; i++) {
		// open per-cpu file
		sprintf(filename, "%s%d", filename_base, i);
		fd = open(filename, O_RDONLY);
		if (fd < 0) {
			printf("fopen() failed: %s\n", strerror(errno));
			return 1;
		}
		// read per-cpu file
		bytesread = read(fd, buf, MAX_BUFLEN);
		while(bytesread > 0) {
			buf[bytesread] = '\0';
			puts(buf);
			bytesread = read(fd, buf, MAX_BUFLEN);
		};
		// close per-cpu file
		if(fd > 0) {
			close(fd);
			fd = 0;
		}
	}
	if(umount("/mnt/relay") && (errno != EINVAL)) {
		printf("umount() failed: %s\n", strerror(errno));
		return 1;
	}
	return 0;
}
```

通过上述实例演示了使用 relay 的过程，虽然上述代码并没有实际用处，但是形象地描述了从用户空间和内核空间两个方面使用 relay 的基本流程。实际应用中对 relay 的使用当然要复杂得多，有关更多用法的演示实例请读者参考 relay 的主页。

2.9　编写 JNI 方法

在 Android 系统中，通过编写 JNI 方法的方式在应用程序框架层提供 Java 接口访问硬件。当为 Android



系统的硬件编写驱动程序时，包括了在 Linux 内核空间实现内核驱动程序和在用户空间实现硬件抽象层接口的工作。实现这两者的目的是为了向更上一层提供硬件访问接口，即为 Android 的 Application Frameworks 层提供硬件服务。众所周知，Android 系统的应用程序是用 Java 语言编写的，而硬件驱动程序是用 C/C++语言来实现的，那么 Java 接口如何去访问 C/C++接口呢？众所周知，Java 提供了 JNI 方法调用，同样在 Android 系统中，Java 应用程序通过 JNI 来调用硬件抽象层接口。

由此可见，JNI 中作为连接顶层 Java 应用程序和底层 C/C++驱动的桥梁。在 Android 底层驱动开发和移植的过程中，JNI 方法的开发工作十分重要。在下面的内容中，将以具体演示例子来介绍为 Android 硬件抽象层接口编写 JNI 方法的过程，以便使得上层的 Java 应用程序能够使用下层提供的硬件服务。

（1）打开 frameworks/base/services/jni 目录，新建文件 com_Android_server_HelloService.cpp，具体实现代码如下所示。

```
#include "jni.h"
#include "JNIHelp.h"
#include <android_runtime/AndroidRuntime.h>
#include <utils/misc.h>
#include <utils/Log.h>
#include <hardware/hardware.h>
#include <hardware/hello.h>
#include <stdio.h>

#include <android/log.h>

#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR,"hello_stub",__VA_ARGS__)
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO,"hello_stub",__VA_ARGS__)

#define LOG_TAG "HelloService"

namespace Android
{
    /*在硬件抽象层中定义的硬件访问结构体，参考<hardware/hello.h>*/
    struct hello_device_t* hello_device = NULL;
    /*通过硬件抽象层定义的硬件访问接口设置硬件寄存器 val 的值*/
    static void hello_setVal(JNIEnv* env, jobject clazz, jint value)
    {
        int val = value;
        LOGI("Hello JNI: set value %d to device.", val);
        if(!hello_device)
        {
            LOGI("Hello JNI: device is not open.");
            return;
        }

        hello_device->set_val(hello_device, val);
    }

    /*通过硬件抽象层定义的硬件访问接口读取硬件寄存器 val 的值*/
    static jint hello_getVal(JNIEnv* env, jobject clazz) {
            int val = 0;
            if(!hello_device)
```

```
            {
                LOGI("Hello JNI: device is not open.");
                return val;
            }
            hello_device->get_val(hello_device, &val);

            LOGI("Hello JNI: get value %d from device.", val);

            return val;
    }

    /*通过硬件抽象层定义的硬件模块打开接口打开硬件设备*/
    static inline int hello_device_open(const hw_module_t* module, struct hello_device_t** device)
    {
        return module->methods->open(module, HELLO_HARDWARE_MODULE_ID, (struct hw_device_t**)
device);
    }


    /*通过硬件模块 ID 来加载指定的硬件抽象层模块并打开硬件*/
    static jboolean hello_init(JNIEnv* env, jclass clazz)
    {
        hello_module_t* module;

        LOGI("Hello JNI: initializing......");

            if(hw_get_module(HELLO_HARDWARE_MODULE_ID, (const struct hw_module_t**)&module) == 0)
            {
                LOGI("Hello JNI: hello Stub found.");
                if(hello_device_open(&(module->common), &hello_device) == 0)
                {
                    LOGI("Hello JNI: hello device is open.");
                    return 0;
                }

                LOGE("Hello JNI: failed to open hello device.");
                return -1;
            }
            LOGE("Hello JNI: failed to get hello stub module.");
            return -1;
        }

          /*JNI 方法表*/
        static const JNINativeMethod method_table[] =
        {
            {"init_native", "()Z", (void*)hello_init},
            {"setVal_native", "(I)V", (void*)hello_setVal},
            {"getVal_native", "()I", (void*)hello_getVal},
        };
```

```
        /*注册 JNI 方法*/
        int register_Android_server_HelloService(JNIEnv *env)
        {
            return jniRegisterNativeMethods(env, "com/Android/server/HelloService", method_table, NELEM
(method_table));
        }

};
```

（2）修改同目录下的文件 onload.cpp，首先在 namespace Android 增加对函数 register_android_server_HelloService 的声明，具体实现代码如下所示。

```
namespace Android {
int register_Android_server_HelloService(JNIEnv *env);
};
```

在 JNI_onLoad 中增加对函数 register_Android_server_HelloService 的调用，具体实现代码如下所示。

```
extern "C" jint JNI_onLoad(JavaVM* vm, void* reserved)
{
register_android_server_HelloService(JNIEnv *env);
}
```

（3）修改同目录下的 Android.mk 文件，在变量 LOCAL_SRC_FILES 中增加如下所示的代码行。

```
LOCAL_SRC_FILES:= /
com_android_server_AlarmManagerService.cpp /
com_android_server_BatteryService.cpp /
com_android_server_InputManager.cpp /
com_android_server_LightsService.cpp /
com_android_server_PowerManagerService.cpp /
com_android_server_SystemServer.cpp /
com_android_server_UsbService.cpp /
com_android_server_VibratorService.cpp /
com_android_server_location_GpsLocationProvider.cpp /
com_android_server_HelloService.cpp /
onload.cpp
```

第 3 章　主流内核系统解析

Android 系统从诞生到现在，得到了市面中各大主流硬件厂商的支持，例如高通（MSM）、德州仪器（OMAP）和联发科都为 Android 设备提供了良好的处理器产品。另外，Android 系统本身也拥有一个虚拟的处理器 Goldfish。对于广大底层和驱动程序开发人员来说，其开发过程离不开上述 3 大主流处理器。为此，本章将依次讲解当今市面中主流处理器平台的内核驱动的架构知识。

3.1　Goldfish 内核和驱动解析

Android 系统的驱动分为专用驱动和设备驱动两大类，其中专用驱动不是 Linux 内核中的标准内容，而是与体系结构和硬件平台无关的纯软件。Android 的专用驱动和 Linux 中的内存驱动类似，主要被保存在 drivers/staging/android 目录中，只有极少数的驱动被保存在其他目录中。在移植 Android 专用驱动时，无须作出任何更改即可进行配置，并可以灵活选择是否使用驱动程序。

其中 Android 中专用驱动的具体类别结构如图 3-1 所示。

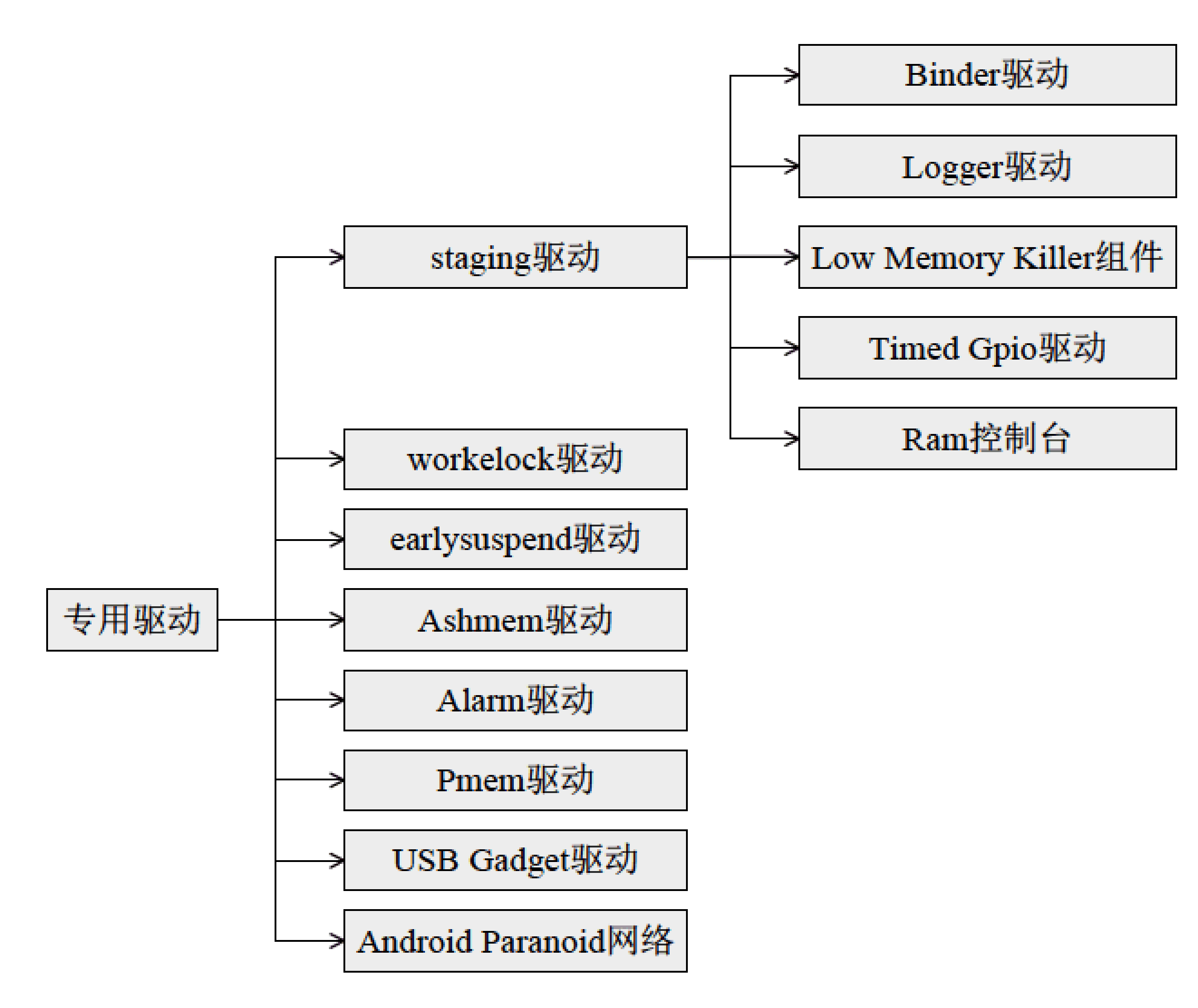


图 3-1　Android 专用驱动的类别结构

在 Android 系统中，专用驱动程序主要被保存在 drivers/staging/android 目录中，此目录是 Android 系统特有的目录，里面还包含了常用的 Kconfig 文件和 Makefile 文件。其中 Makefile 的内容如下所示。

```
obj-$(CONFIG_ANDROID_BINDER_IPC) += binder.o
obj-$(CONFIG_ANDROID_LOGGER) += logger.o
obj-$(CONFIG_ANDROID_RAM_CONSOLE) += ram_console.o
obj-$(CONFIG_ANDROID_TIMED_OUTPUT) += timed_output.o
obj-$(CONFIG_ANDROID_TIMED_GPIO) += timed_gpio.o
obj-$(CONFIG_ANDROID_LOW_MEMORY_KILLER) += lowmemorykiller.o
```

对于上述内容的具体说明如下。

☑　binder 和 logger：是两个普通的 misc 驱动程序。
☑　timed_output：是一种 Android 特有的驱动程序框架。
☑　timed_gpio：是基于 timed_output 的一个驱动程序。
☑　lowmemorykiller：是一个内存管理的组件。
☑　ram_console：是一个利用控制台驱动的框架。

注意：在获取的Android源码中，已经内置了虚拟处理器Goldfish，而MSM和OMAP处理厂商的内核和驱动程序都是以Goldfish作为参考模型进行开发的，所以说Goldfish是我们学习本书内容的根本，本书后面的内容页将以Goldfish的内容为主。

3.1.1　Goldfish 基础

Goldfish 是谷歌公司为 Android 推出的一种虚拟的 ARM 处理器，在 Android 的仿真环境中使用，Android 模拟器通过运行 Goldfish 来运行 arm926t 指令集。其中 arm926t 属于 armv5 构架，Goldfish 处理器有 ARMv5 和 ARMv7 两个版本，在一般情况下只需使用 ARMv5 的版本即可。

Android 模拟器的 kernel 是虚构出来的 ARM，CPU 名为 goldfish，可能很多读者下载的源码中没有 Goldfish 内核的源码，此时需要先用 git 命令下载 Goldfish 的 sourcecode 代码包。

```
git clone https://android.googlesource.com/kernel/goldfish.git
```

或者：

```
$ git clone git://android.git.kernel.org/kernel/common.git
```

然后可以用以下命令选择指定的版本并复制代码。

```
$cd goldfish
$git branch -a
* (no branch)
master
remotes/origin/HEAD -> origin/master
remotes/origin/android-goldfish-2.6.29
remotes/origin/master
$git checkout remotes/origin/android-goldfish-2.6.29 -b goldfish
```

获取到 Goldfish 内核代码后，可以在 Android 的模拟器中使用编译生成的 Linux 内核镜像。在启动模拟器时，Linux Kernel 镜像默认使用如下文件。

```
prebuilt/android-arm/kernel/kernel-qemu
```

在 Linux 的内核中，Goldfish 作为 ARM 体系结构的一种 mach，它的核心内容被保存在 arch/arm/mach-goldfish 目录中。

文件 goldfish_defconfig 被保存在 kernel/arch/arm/configs/目录中。

在文件 goldfish_defconfig 中定义了与 Android 系统相关的宏，主要实现代码如下所示。

```
# android
#
CONFIG_ANDROID=y
CONFIG_ANDROID_BUNDER_IPC=y          #binder ipc 驱动程序
CONFIG_ANDROID_LOGGER=y              #log 记录器驱动程序
# CONFIG_ANDROID_RAM_CONSOLE is not set
CONFIG_ANDROID_TIMED_OUTPUT=y        #定时输出驱动程序框架
CONFIG_ANDROID_LOW_MEMORY_KILLER=y
CONFIG_ANDROID_PMEM=y                #物理内存驱动程序
CONFIG_ASHMEM=y                      #匿名共享内存驱动程序
CONFIG_RTC_INTF_ALARM=y
CONFIG_HAS_WAKELOCK=y                #电源管理相关的部分 wakelock 和 earlysuspend
CONFIG_HAS_EARLYSUSPEND=y
CONFIG_WAKELOCK=y
CONFIG_WAKELOCK_STAT=y
CONFIG_USER_WAKELOCK=y
CONFIG_EARLYSUSPEND=y
```

另外也定义了处理器虚拟设备的驱动程序，具体代码如下所示。

```
CONFIG_MTD_GOLDFISH_NAND=y
CONFIG_KEYBOARD_GOLDFISH_EVENTS=y
CONFIG_GOLDFISH_TTY=y
CONFIG_BATTERY_GOLDFISH=y
CONFIG_FB_GOLDFISH=y
CONFIG_MMC_GOLDFISH=y
CONFIG_RTC_DRV_GOLDFISH=y
```

在 Goldfish 处理器中，无论是各个配置选项的体系结构还是 Goldfish 的虚拟驱动程序，都基于标准 Linux 的内容的驱动程序框架，但是在不同的硬件平台中移植这些设备的方式不同。Android 专用的驱动程序是 Android 系统中特有的内容，并不符合 Linux 标准要求，但是和硬件平台无关。

在 Android 的发展过程中，Goldfish 内核的版本也从 Linux 2.6.25 升级到了 Linux 3.18，并且从 Linux 3.9 内核开始，开始全面支持 Goldfish 模拟器。Goldfish 处理器内核的 Linux 内核和标准的 Linux 内核的差别如下。

☑ Goldfish 机器的移植。
☑ Goldfish 一些虚拟设备的驱动程序。
☑ Android 中特有的驱动程序和组件。

在 Linux 源代码的根目录中，配置并编译 Goldfish 内核的命令如下所示。

```
$make ARCH=arm goldfish_defconfig .config
$make ARCH=arm CROSS_COMPILE={path}/arm-none-linux-gnueabi-
```

其中，“CROSS_COMPILE=”用于指定交叉编译工具的路径。

编译之后会输出：

```
LD vmlinux
SYSMAP system.map
SYSMAP .tmp_system.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS arch/arm/boot/compressed/head.o
GZIP arch/arm/boot/compressed/piggy.gz
AS arch/arm/boot/compressed/piggy.o
CC arch/arm/boot/compressed/misc.o
LD arch/arm/boot/compressed/vmlinux
    OBJCONPY arch/arm/boot/zImage
    Kernel: arch/arm/boot/zImage is ready
```

在上述输出结果中，vmlinux 是 Linux 进行编译和链接之后生成的 Elf 格式的文件，Image 是未经过压缩的二进制文件，piggy 是一个解压缩程序，zImage 是解压缩程序和压缩内核的组合。

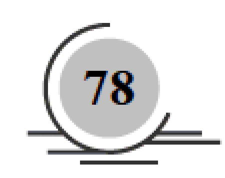
在 Android 源代码的根目录中，vmlinux 和 zImage 分别对应 Android 代码 prebuilt 中的预编译的 arm 内核。

3.1.2 Logger 驱动

Logger 驱动是 Android 系统的专用驱动，这是一个轻量级的 log 驱动，通常被作为一个工具来使用，功能是为用户层程序提供 Log 支持。Logger 驱动有如下 3 个设备节点。

☑ /dev/log/main：主要的 log。
☑ /dev/log/event：事件的 log。
☑ /dev/log/radio：Modem 部分的 log。

Logger 驱动为用户空间提供了 ioctl 接口、read 接口和异步 write 接口，其主设备号为 10（Misc Driver），其实现源代码位于 kernel/include/linux/logger.h 和 kernel/drivers/misc/logger.c 源文件中。



对于非本用户或本组来说，Logger 驱动程序的设备节点是可写的，不可读的。在 Android 用户空间中，使用库 liblog 封装了 Logger 驱动程序，其保存路径为 system/core/liblog，并通过 logcat 程序调用 Logger 驱动。logcat 程序是一个可知性程序，当用户取出系统 log 信息后，会在系统中使用 logcat 程序作为辅助工具，logcat 程序的代码路径为 system/core/logcat。

3.1.3　Low Memory Killer 组件

内存管理组件 Low Memory Killer 是 Android 系统的专用驱动，功能是根据需要释放内存的需要而杀死某一个进程。因为毕竟移动设备没有 PC 机那么强大，所以需要随时优化进程。Low Memory Killer 机制十分灵活，当内存不够时会试图结束一个进程。组件 Low Memory Killer 通过调用 Linux 内存管理系统接口的方式来注册一个 shrinker，此处的 shrinker 是通过 Low Memory Killer 实现的。

组件 Low Memory Killer 的源代码位于 drivers/staging/android /lowmemorykiller.c 文件中。

文件 lowmemorykiller.c 的核心代码如下所示。

```
static struct shrinker lowmem_shrinker = {
	.shrink = lowmem_shrink,
	.seeks = DEFAULT_SEEKS * 16
};
module_param_named(cost, lowmem_shrinker.seeks, int, S_IRUGO | S_IWUSR);
module_param_array_named(adj, lowmem_adj, int, &lowmem_adj_size,
			 S_IRUGO | S_IWUSR);
module_param_array_named(minfree, lowmem_minfree, uint, &lowmem_minfree_size,
			 S_IRUGO | S_IWUSR);
module_param_named(debug_level, lowmem_debug_level, uint, S_IRUGO | S_IWUSR);

module_init(lowmem_init);
module_exit(lowmem_exit);

MODULE_LICENSE("GPL");
```

有 /sys/module/lowmemorykiller/parameters/adj 和 /sys/module/lowmemorykiller/parameters/minfree 两个与组件 Low Memory Killer 相关的配置文件，在里面定义了系统配置的相关参数。

标准 Linux 内核 OOM Killer 在 mm/oom_kill.c 中实现，在 mm/page_alloc.a_alloc_pages_may_oom 中被调用。文件 oom_kill.c 最主要的函数是 out_of_memory()，它选择一个 bad 进程通过发送 SIGKILL 信号来杀死进程。

在 out_of_memory 中通过调用 select_bad_process 选择杀死一个进程，选择的依据在 badness()函数中实现，基于多个标准来给每个进程算分，分最高的被选中杀死。基本上是占用内存越多，oom_adj 越大，越有可能被选中。

由此可以看出，Android 的 Low Memory Killer 和标准的 OOM Killer 的很多思路是一致的，只不过 Low Memory Killer 作为一个 shrinker 实现；而 OOM Killer 则在分配内存时被调用（如果内存资源很紧张）。Android 的 Low Memory Killer 的实现较为简洁，这点从代码尺寸就能看到，但并不比 OOM Killer 更为灵活，它只不过是另一种 OOM Killer。

3.1.4　Timed Output 驱动

Timed Output 驱动是 Android 系统中一个很重要的专有驱动框架，例如 Timed Output 驱动程序框架可以实现 Vibrator（振动）功能的驱动程序。Timed Output 驱动是基于 sys 文件系统来完成的，能够对设备进行定时控制功能，目前支持设备有 Vibrator（振动）和 LED（闪光灯）设备。

Timed Output 驱动会注册 sys/class/timed_output/目录，每一个注册都会实现一个 Timed Output 设备，例如 Vibrator 和 LED。这样将会在 sys/class/timed_output/目录下新建一个和设备同名的子目录，在子目录下通过对 enable 文件的读写实现对设备的控制和显示。

Timed Output 驱动有 drivers/staging/android/timed_output.c 和 drivers/staging/android/timed_output.h 两个实现文件。

3.1.5 Timed Gpio 驱动

Timed Gpio 驱动是 Android 系统的一个专有驱动，是基于 Timed_Output 驱动的一个驱动程序，能够定时控制 GPIO。Timed Gpio 可以调用 Timed Output 框架注册一个驱动程序，其驱动程序保存在 drivers/staging/android/timed_gpio.h 和 drivers/staging/android/timed_gpio.c 两个文件中。

在文件 timed_gpio.h 中定义了 Timed Gpio 驱动的名称，并设置结构体 timed_gpio 作为驱动的私有结构体。文件 timed_gpio.h 的实现代码如下所示。

```
#ifndef _LINUX_TIMED_GPIO_H
#define _LINUX_TIMED_GPIO_H
#define TIMED_GPIO_NAME "timed-gpio"//定义 Timed Gpio 驱动的名称
struct timed_gpio {
	const char *name;
	unsigned  gpio;
	int max_timeout;
	u8 active_low;
};
struct timed_gpio_platform_data {
	int num_gpios;
	struct timed_gpio *gpios;
};
#endif
```

在文件 timed_gpio.c 中，通过下面的两个函数分别实现对驱动设备的注册和注销。

```
int timed_output_dev_register(struct timed_output_dev *tdev)
void timed_output_dev_unregister(struct timed_output_dev *tdev)
```

3.1.6 Ram Console 驱动

在 Android 平台中，Ram Console 驱动提供了一种可以辅助调试的内核机制，是一个控制台驱动的框架。为了给 Android 提供调试功能，可以将调试日志信息写入 Ram Console 的设备中，这是一个基于 RAM 的 Buffer（缓存）设备。

Ram Console 与用户空间之间的接口是 proc 文件系统，在 proc 中使用 last_kmsg 文件来表示 kernel 最后输出的信息。在 Android 平台中，在 drivers/staging/android/ram_console.c 文件中实现 Ram Console 驱动程序。

在文件 ram_console.c 中实现注册功能的函数代码如下所示。

```
static int __init ram_console_init(struct ram_console_buffer *buffer,
				size_t buffer_size, char *old_buf)
{
#ifdef CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION
	int numerr;
	uint8_t *par;
#endif
	ram_console_buffer = buffer;
```

```
ram_console_buffer_size =
        buffer_size - sizeof(struct ram_console_buffer);

if (ram_console_buffer_size > buffer_size) {
        pr_err("ram_console: buffer %p, invalid size %zu, "
                "datasize %zu\n", buffer, buffer_size,
                ram_console_buffer_size);
        return 0;
}
```

3.1.7　Ashmem 驱动

Ashmem 是 Android 的内存分配/共享机制，经常被称为匿名共享内存。在 dev 目录下对应的设备是/dev/ashmem。和 malloc、anonymous/named mmap 等传统的内存分配机制相比，Ashmem 的好处是提供了辅助内核内存回收算法的 pin/unpin 机制。

Ashmem 基于 MMAP 系统调用，不同的进程可以将同一段物理内存映射到各自的虚拟地址控制以实现共享。Ashmem 与 MMAP 的不同之处是，Ashmem 与 Cache Shrinker 相互关联，可以在适当时机回收这些共享内存，而 MMAP 则不具备这个功能。

Ashmem 的源代码保存在内存管理的/mm 目录中，其具体实现文件是 android/mydroid/kernel/mm/ ashmem.c。

Ashmem 的头文件是 include/linux/ashmem.h。

3.1.8　Pmem 驱动

在 Android 系统中，Pmem 驱动和 Ashmem 驱动都是通过 MMAP 实现共享功能的，两者的区别是 Pmem 的共享区域是一段连续的物理内存，而 Ashmem 在虚拟空间是连续的共享区域，在物理内存中并不一定连续。

Pmem 的源代码在文件 drivers/misc/pmem.c 中实现，依赖于 Linux 的 misc device 和 platform driver 框架。在一个系统中可以有多个 Pmem 驱动，默认值是最多 10 个。

在初始化 Pmem 模块时会注册一个 platform driver，在后面的 probe 操作时会创建 MISC 设备文件来分配内存，并完成初始化工作。

在 Android 系统中，Pmem 通过如下结构体来维护分配的共享内存。

- ☑ pmem_info：代表一个 Pmem 设备分配的内存块。
- ☑ pmem_data：代表该内存块的一个子块，是分配的基本单位。每当应用层要分配一块 Pmem 内存时，就会有一个 pmem_data 来表示这个被分配的内存块。
- ☑ pmem_region：负责把每个子块分成多个区域。

在进行 open 操作时，并不是打开一个 pmem_info 表示的整个 Pmem 内存块，而是创建一个 pmem_data 以备使用。一个应用可以通过 ioctl 来分配 pmem_data 中的一个区域，并可以把它 map（映射）到进程空间；并不一定每次都要分配和 map 整个 pmem_data 内存块。

3.1.9　Alarm 驱动

在 Android 系统中，Alarm 是一个能够提供定时器功能的硬件时钟，用于把设备从睡眠状态唤醒，并且同时也提供了一个在设备睡眠时仍然会运行的时钟基准。在应用层上，有关时间的应用都需要 Alarm 的支持，源代码位于 drivers/rtc/alarm.c 文件中。

Alarm 的设备名为/dev/alarm，打开源码后首先看到如下包含代码。

```
include<linux /android_alarm.h>
```

在里面定义了一些和 Alarm 相关的信息，主要包括如下 5 种类型的 Alarm。

☑ _WAKEUP 类型：表示在触发 Alarm 时需要唤醒设备，反之则不需要唤醒设备。
☑ ANDROID_ALARM_RTC 类型：表示在指定的某一时刻触发 Alarm。
☑ ANDROID_ALARM_ELAPSED_REALTIME 类型：表示在设备启动后，流逝的时间达到总时间之后触发 Alarm。
☑ ANDROID_ALARM_SYSTEMTIME 类型：表示系统时间。
☑ ANDROID_ALARM_TYPE_COUNT 类型：表示 Alram 类型的计数。

Alarm 返回标记随着 Alarm 的类型而改变。通过定义的宏实现禁用 Alarm、Alarm 等待、设置 Alarm 等功能。

3.1.10 USB Gadget 驱动

USB Gadget 是 Linux 系统中的 USB 驱动程序，在 Android 系统中，新增了 ADB Garget 驱动来实现 USB 驱动功能。当使用 Garget 驱动时，Android 将作为一个 USB 设备而提供一个 ADB 接口。

在 Linux 系统中，ADB Garget 的功能主要体现在设备端，并且每一个硬件只能选一个。在 ADB Garget 中包含了 ADB 的调试功能和大容量存储器的功能。

ADB Garget 驱动程序的源码保存在/drivers/usb/gadget 目录下，分别通过文件 android.c、f_adb.c 和 f_mass_storage.c 来实现。其中 g_android.ko 是由这 3 个文件编译而来，文件 android.c 依赖于 f_adb.c 和 f_mass_storage.c（文件 f_adb.c 和文件 f_mass_storage.c 之间没有依赖关系）。f_adb.c 是实现 ADB 功能的文件，f_mass_storage.c 是标准的文件，包含此文件的目的是为了同时实现大容量存储器的功能。

在文件 android.c 中注册了一个 MISC 设备 dev/android_adb_enable，当打开这个设备时表示用 ADB Garget 的功能。在文件 android.c 中需要分别注册 adb 和 mass storage，具体实现代码如下所示。

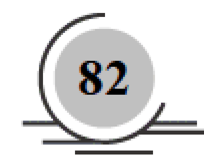
```
static int __init android_bind_config(struct usb_configuration *c)
        {
            struct android_dev *dev = _android_dev;
            int ret;
            printk(KERN_DEBUG "android_bind_config\n");
            ret = mass_storage_function_add(dev->cdev, c, dev->nluns);
            if (ret)
                return ret;
            return adb_function_add(dev->cdev, c);
}
```

在文件 f_afb.c 中也注册了一个 MISC 设备 dev/android_adb，此设备支持读写功能。

3.1.11 Paranoid 驱动介绍

Paranoid 是 Android 系统中的网络驱动程序，Android 对 Linux 内核的网络部分进行了改动，通过改动后增加了网络认证机制。上述改动功能是通过宏 ANDROID_PARANOID_NETWORK 实现的，在修改中涉及了 Linux 源码中的以下文件。

☑ net/ipv4/af_inet.c：IPV4 协议文件。
☑ net/ipv6/af_inet3.c：IPV6 协议文件。
☑ net/bluetooth/af_bluetooth.c：蓝牙协议文件。
☑ security/commoncap.c：安全性文件。



在上述文件中，前 3 个是 3 种不同网络协议中处理协议方面的文件，在逻辑上是并列的关系。有关网络部分 AID 的定义是在文件 include/linux/android_aid.h 中实现的，对应代码如下所示。

```
#ifndef _LINUX_ANDROID_AID_H
#define _LINUX_ANDROID_AID_H
/* AIDs that the kernel treats differently */
#define AID_NET_BT_ADMIN 3001
#define AID_NET_BT 3002
#define AID_INET 3003
#define AID_NET_RAW 3004
#endif
```

在文件 af_inet.c 中会进一步检查 AID，只有符合时才返回 1，如果没有附加此特性则直接返回 1。在文件 commoncap.c 中与之相关的代码如下所示。

```
int cap_capable(struct task_struct *tsk, const struct cred *cred,
                struct user_namespace *targ_ns, int cap, int audit)
 {
        for (;;) {
                if (targ_ns != &init_user_ns && targ_ns->creator == cred->user)
                        return 0;

                /* 需要必要的能力? */
                if (targ_ns == cred->user->user_ns)
                        return cap_raised(cred->cap_effective, cap) ? 0 : -EPERM;

                /*尝试了所有父母的 namespaces? */
                if (targ_ns == &init_user_ns)
                        return -EPERM;
                targ_ns = targ_ns->creator->user_ns;
        }
}
```

通过上述代码实现了对 AID 的判断，如果 AID 符合要求则返回 0，并不会再使用函数 return cap_raised() 进行处理。

3.1.12 Goldfish 的设备驱动

Android 专用驱动的内容已经讲解完毕，下面将介绍 Goldfish 平台中设备驱动的基本知识，Goldfish 处理平台中的设备驱动的类别结构如图 3-2 所示。

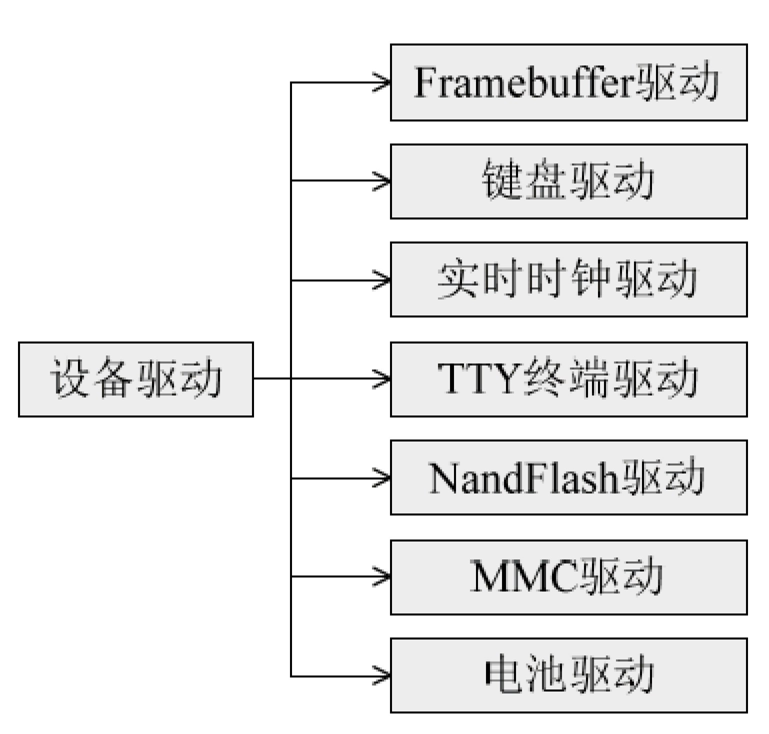


图 3-2 设备驱动类别结构

（1）Framebuffer 驱动

在 Android 中使用 SurfaceFlinger 作为屏幕合成引擎，用它管理来自各个窗口的 Surface objects，然后将其写入 Framebuffer 中。每一个 Surface 都是双缓冲的，SurfaceFlinger 使用前 buffer 负责合成，使用后 buffer 负责绘制。一旦绘制完成，Android 通过页翻转操作，交换 Y 轴坐标的偏移量，选择不同 buffer。在 EGL 显示服务初始化时，如果虚拟 Y 轴分辨率大于实际 Y 轴分辨率，说明 Framebuffer 可以直接使用双缓冲。否则，后 buffer 要复制到前 buffer，这样会导致页交换延迟。为了提高系统性能，Framebuffer 驱动最好提供双缓冲机制。

Goldfish 的 Framebuffer 驱动中提供了双缓冲机制，用于 AndroidSDK 中基于 QEMU 的模拟器。

Framebuffer 对应的源文件保存在 linux/drivers/video/目录下。总体抽象设备文件为 fbcon.c，在此目录下还有与各种显卡驱动相关的源文件。Framebuffer 设备驱动基于下面的文件。

- ☑ linux/include/linux/fb.h：定义一些变量结构和宏。
- ☑ linux/drivers/video/fbmem.c：实现设备入口和初始化。
- ☑ xxxfb.c：自己添加的设备驱动文件，例如 struct fb_info，有两个实现入口点函数，分别是 xxxfb_init()和 xxxfb_setup()。

（2）键盘驱动

Goldfish 平台中的键盘驱动是 Goldfish_events，其源代码路径为 drivers/input/keyboard/goldfish_events.c。在文件 goldfish_events.c 先定义了枚举，具体代码如下所示。

```
enum {
	REG_READ = 0x00,
	REG_SET_PAGE = 0x00,
	REG_LEN = 0x04,
	REG_DATA = 0x08,
	PAGE_NAME = 0x00000,
	PAGE_EVBITS = 0x10000,
	PAGE_ABSDATA = 0x20000 | EV_ABS,
};
```

然后定义了数据结构 event_dev，具体代码如下所示。

```
struct event_dev {
	struct input_dev *input;
	int irq;
	unsigned addr;
	char name[0];
};
```

然后进行模块初始化，对应代码如下所示。

```
module_init(events_init);
static int __devinit events_init(void){
return platform_driver_register(&events_driver);
}
```

通过初始化实现注册功能，对应代码如下所示。

```
static struct platform_driver events_driver = {
	.probe = events_probe,
	.driver = {
		.name = "goldfish_events",
	},
```

在函数 platform_driver_register()中会执行下面的代码。

```
drv->driver.bus = &platform_bus_type;
if (drv->probe)
drv->driver.probe = platform_drv_probe;
```

最后调用到函数 platform_drv_probe()，此函数的实现代码如下所示。

```
static int platform_drv_probe(struct device *_dev){
struct platform_driver *drv = to_platform_driver(_dev->driver)
struct platform_device *dev = to_platform_device(_dev);
return drv->probe(dev);
}
```

在上述函数代码中，to_platform_driver(_dev->driver)的作用是返回一个 platform_driver 型的指针，而 to_platform_device(_dev)的作用是返回一个 platform_device 的指针。

（3）实时时钟驱动

Goldfish 平台中的实时时钟驱动就是 RTC 设备，这也是 Linux 中的一种标准驱动程序，在用户空间提供了设备节点，例如 MISC 和自定义字符设备，其源代码路径为 kernel/drivers/rtc/rtc-goldfish.c。

其中 goldfish_rtc_read_time()是其读取时间的调用函数。

（4）TTY 终端驱动

在 Goldfish 平台中，TTY 终端驱动提供了虚拟串口功能，其实现源代码被保存在 drivers/char/goldfish_tty.c 文件中。

Goldfish 的 TTY 中断驱动程序在用户空间有 3 个设备，对应的节点分别是 dev/ttyS0、dev/ttyS1 和 dev/ttyS2。TTY 终端驱动只支持写操作，驱动程序写功能是通过文件 goldfish_tty.c 的 goldfish_tty_ do_write()函数实现的。

（5）NandFlash 驱动

在 Goldfish 平台中，NandFlash 驱动提供了对 Flash 设备的支持，其实现源代码被保存在 kernel/drivers/mtd/devices/goldfish_nand.c 和 kernel/drivers/mtd/devices/goldfish_nand_reg.h 文件中。

NandFlash 驱动程序是标准的 MTD 驱动程序，所以 Goldfish 的 Nand 驱动程序将会为每一个分区构建字符设备和块设备。在同一个分区中，可能会有两个字符设备分别用于读写操作和只读操作。

（6）MMC 驱动

在 Goldfish 平台中，MMC 驱动程序是标准的 MMC 主机驱动程序，在手机应用中常用于实现 SD 卡驱动，有时也被称为多媒体驱动。MMC 驱动的标准实现源代码被保存在 kernel/drivers/mmc/host/goldfish.c 文件中。

当有 MMC 或者 SD 卡注册时，才会使用 MMC 驱动程序。

（7）电池驱动

Goldfish 平台中的电池驱动的标准实现源代码被保存在 kernel/drivers/power/goldfish_battery.c 文件中。

这里的电池驱动是一个 power_supply 驱动程序，能够读取电池设备的电量属性，例如剩余电量和总电量等。获取属性功能是通过函数 goldfish_ac_get_property()实现的。

3.2　MSM 内核和驱动架构

在 3.1 节已详细讲解了 Goldfish 内核移植和驱动的基本知识，本节将简要介绍 MSM 内核的基本知识，并简要讲解内核移植和各种驱动的基本知识。

3.2.1　高通公司介绍

MSM 是美国高通公司的处理器产品，是 Android 系统最常用的处理器产品之一。美国高通公司以其 CDMA（码分多址）数字技术为基础，开发并提供富于创意的数字无线通信产品和服务。如今，美国高通公司正积极倡导全球快速部署 3G 网络、手机及应用。

高通公司总部驻于美国加利福尼亚州圣迭戈市，高通公司的股票是标准普尔 500 指数的成分股，公司业务涵盖技术领先的 3G、4G 芯片组，系统软件以及开发工具和产品，技术许可的授予，BREW 应用开发平台，QChat、BREWChatVoIP 解决方案技术，QPoint 定位解决方案，Eudora 电子邮件软件，包括双向数据通信系统、无线咨询及网络管理服务等的全面无线解决方案，MediaFLO 系统和 GSM1x 技术等。美国高通公司拥有所有 3000 多项 CDMA 及其他技术的专利及专利申请，这些标准已经被全球制定标准机构普遍采纳或建议采纳。高通已经向全球 125 家以上电信设备制造商发放了 CDMA 专利许可。

作为一项新兴技术，CDMA 正迅速风靡全球并已占据 20%的无线市场。目前，全球 CDMA 用户已超过 2.56 亿，遍布 70 个国家的 156 家运营商已经商用 3GCDMA 业务。2002 年，高通公司芯片销售创历史佳绩；

1994 年至今，高通公司已向全球包括中国在内的众多制造商提供了累计超过 15 亿枚芯片。

3.2.2 常见的 MSM 处理器产品

（1）MSM7200

MSM7200 解决方案支持上行密集型（uplink-intensive）服务，例如 IP 语音（VoIP）、3D 多人无线游戏以及实时共享高质量视频和图像的一按式多媒体（push-to-multimedia）应用。此外，MSM7200 芯片组还支持大容量附件电子邮件的发送和接收，从而进一步提高企业效率。

MSM7200 芯片组支持的下行链路的数据传输速率高达 7.2Mbps，上行链路的数据传输速率高达 5.76Mbps，这一速率高于有线宽带连接的速率。作为融合平台的一部分，MSM7200 还支持第三方操作系统，从而进一步将消费类电子产品功能和无线通信功能融合在一起。

高通 MSM7200 芯片的 CPU 部分主频高达 400MHz，采用双核构架，有一个 400MHz 的 ARM11 核心负责程序部分，一个频率为 274MHz 的 ARM9 核心负责通信，拥有高速的网络接口，可以支持 GPRS、EDGE、WCDMA、HSDPA、HSUPA 等数据连接，另外 MSM7200 还可以提供 Java 硬件加速、拥有独立的音频处理模块、内建 Q3Dimension 3D 渲染引擎，支持 OpenGL ES 3D 图形加速，拥有每秒 400 万多边形计算、133 万像素填充能力。从硬件上支持 H.263 以及 H.264 的视频解码。在摄像头方面最大可以支持并且还内建 GPS 模块。可以说 MSM 是一块高度集成的处理器，而且性能非常强劲。

（2）MSM7201A

MSM7201A 是单芯片、双核的解决方案，可以提供高速数据处理功能、硬件加速多媒体功能、3D 图形以及嵌入式多模 3G 移动宽带连接以实现完美的无线体验。

MSM7201A 芯片组内建 3D 图形处理模块以及嵌入模式的 3G 连接，还具备高速数据传输以及处理功能，同时支持硬件加速技术。这样的硬件设计丰富了 Android 平台的功能，支持多样化的应用服务，为用户带来新鲜的个性体验。这个主频为 528MHz 的 MSM7201A 芯片组已经在 HTC Touch Diamond 和 Touch Pro 这两款机器上使用了。

MSM7201A 芯片组支持高分辨率的图像以及视频播放，流媒体功能表现也很出色，支持包括 YouTube 在内的服务。300 万像素的摄像头可以有效地扫描条形码，用户可以在网上查找到相关物品的售价，这样方便比较同样商品的售价。SM7201A 芯片组支持 GPS 卫星定位功能。而且高通公司透露 MSM7201A 芯片组将会在未来其他制造商的 Android 平台手机上使用。

（3）QSD8250

QSD8250 支持 HSPA 数据传输，下行速率可达 7.2Mbps，上行速率达 5.76Mbps，并提供全向后兼容（full backward compatibility）。双模的 QSD8650 支持 HSPA 及 CDMA2000 1xEV-DO Rev.B，并提供全向后兼容。此两款解决方案均含有 1GHz 的微处理器核心，搭配高通第六代以 600MHz 运作的 DSP 核心，可提供随开即用（instant-on）及全时连线（always-connected）的使用者体验。

Snapdragon 支持高传真影像解码、1200 万像素的照相功能、GPS、移动电视（含 MediaFLO、DVB-H 及/或 ISDB-T 标准）、WiFi 及蓝牙功能，可协助装置制造商设计即时、无缝连线的轻薄手机。

（4）QSD8650

QSD8650 和 QSD8250 是高通为不同网络用户而设计的两个芯片解决方案，其中，QSD8650 是双模芯片解决方案，它不仅可以像 QSD8250 那样支持 HSPA 数据传输，而且支持 CDMA2000、CDMA1x 网络以及 EV-DO Rev.B 数据传输。这两个解决方案都包含 1GHz 处理器核心。

（5）高通 MSM8255

MSM8255 采用 45 纳米级单核心技术的 CPU 芯片，制程的提升有助于省电和缩小芯片尺寸，而省电是比较重要的提升。其次，GPU 的提升也非常明显，相比 QSD8250 内建 Adreno 200 图形处理芯片，而 MSM8255

为 Adreno 205，虽然数字只差 5，但是性能翻倍，对于 Android 这样耗 GPU 的系统来说，高性能 GPU 就更有必要了。

MSM8255 是世界首款 1.4GHz 单核，MSM8x55 芯片组平台包括 MSM8255™ 和 MSM8655™，专为高性能智能手机和平板电脑设计，以最新设计和优化的多媒体子系统及 45nm 处理技术为特色，在低能耗的同时提供一流的单核处理性能。这一平台同时包括 APQ8055 处理器，专为平板电脑和大型展示设备设计，无须 WWAN 调制解调器。

CPU（中央处理器）Scorpion 单核，高达 1.4GHzGPU（图形处理器）Adreno™ 205：高级移动图像多媒体高分辨率（720p）视频录像和回放技术，每秒可达 30 帧多音频和视频编解码器支持高分辨率 XGA（1024x768）显示 Dolby® 5.1 环绕声立体 3D 捕捉和回放支持 1200 万像素双摄像头。

（6）MSM8260

MSM8260 是世界首款 1.5GHz 移动异步双核，MSM8x60™ 芯片组平台包括 MSM8260™ 和 MSM8660™，满足多任务、高级游戏和娱乐需要，使用低电能 45nm 处理技术，具有更高的整合度和性能。这一平台同时包括 APQ8060 处理器，专为平板电脑和大型展示设备设计，无须使用 WWAN 调制解调器。

（7）骁龙系列

骁龙是高通公司推出的高度集成的“全合一”移动处理器系列平台，分别覆盖入门级智能手机乃至高端智能手机、平板电脑以及下一代智能终端。Snapdragon 以基于 ARM 架构定制的微处理器内核为基础，结合了业内领先的 3G/4G 移动宽带技术与强大的多媒体功能、3D 图形功能和 GPS 引擎。2012 年 2 月 20 日，高通正式将 Snapdragon 系列处理器的中文名称定为“骁龙”。当前智能机的旗舰机型都是用的晓龙系列的产品，例如 Note 3 和 Galaxy S5 等。

3.2.3 MSM 内核移植

MSM 处理器平台中的 Linux 内核和标准的 Linux 内核相比，具有以下 3 点差别。

☑ MSM 及其板级平台机器的移植。

☑ MSM 及其板级平台一些虚拟设备的驱动程序。

☑ Android 中特有的驱动程序和组件。

在 Android 开源网站上，使用 git 工具可以得到 MSM 内核代码。操作命令如下所示。

```
$ git clone git://android.git.kernel.org/kernel/msm.git
```

在通常情况下，MSM 内核 git 的代码仓库中有多个分支可以选择，例如 origin/android-msm-2.6.29、origin/android-msm-2.6.23-nexusone 和 origin/android-msm-hammerhead-3.3-kk-fr1。

以比较成熟的版本 2.6.29 为例，进行编译的命令如下所示。

```
$ git checkout –b android-msm-2.6.29 origin/android-msm-2.6.29
$ git make ARCH=arm msm_defconfig .config
$ git make ARCH=arm CROSS_COMPILE={path}/arm-none-linux-gnueabi-
```

选择 Nexus One 中使用的 MSM 内核版本，并且进行编译的方式如下所示。

```
$ git checkout –b android-msm-2.6.23-nexusone origin/android-msm-2.6.23-nexusone
$ git make ARCH=arm msm_defconfig .config
$ git make ARCH=arm CROSS_COMPILE={path}/arm-none-linux-gnueabi-
```

在当前应用中，使用 MSM 平台的 Linux 内核主要有如下两种版本。

☑ 针对 MSM7kxx 系列的处理器：config 文件的路径是 arch/arm/configs/msm_defconfig。

☑ 针对 QSD8kxx 系列的处理器（snapdragoh）：config 文件的路径为 arch/arm/configs/mahimahi_defconfig。

上述两个版本使用了不同的 Linux 代码和配置文件。

例如在 Linux 3.4 内核中，MSM 的源码片段如下所示。

```
CONFIG_EXPERIMENTAL=y
CONFIG_IKCONFIG=y
CONFIG_IKCONFIG_PROC=y
CONFIG_BLK_DEV_INITRD=y
CONFIG_SLAB=y
# CONFIG_BLK_DEV_BSG is not set
# CONFIG_IOSCHED_DEADLINE is not set
# CONFIG_IOSCHED_CFQ is not set
CONFIG_ARCH_MSM=y
CONFIG_MACH_HALIBUT=y
CONFIG_NO_HZ=y
CONFIG_HIGH_RES_TIMERS=y
CONFIG_PREEMPT=y
CONFIG_AEABI=y
# CONFIG_OABI_COMPAT is not set
CONFIG_ZBOOT_ROM_TEXT=0x0
CONFIG_ZBOOT_ROM_BSS=0x0
CONFIG_CMDLINE="mem=64M console=ttyMSM,115200n8"
CONFIG_PM=y
CONFIG_NET=y
CONFIG_UNIX=y
CONFIG_INET=y
# CONFIG_INET_XFRM_MODE_TRANSPORT is not set
# CONFIG_INET_XFRM_MODE_TUNNEL is not set
# CONFIG_INET_XFRM_MODE_BEET is not set
# CONFIG_INET_DIAG is not set
# CONFIG_IPV6 is not set
CONFIG_MTD=y
CONFIG_MTD_PARTITIONS=y
CONFIG_MTD_CMDLINE_PARTS=y
CONFIG_MTD_CHAR=y
CONFIG_MTD_BLOCK=y
CONFIG_NETDEVICES=y
```

在 MSM 处理器平台中，Linux 的移植部分内容主要在如下目录中。

☑ arch/arm/mach-msm/：MSM 平台部分移植的核心部分，其中包含了 qdsp5 和 qdsp6 两个目录，它们分别是 5 代 DSP 和 6 代 DSP 在应用处理器端的相关内核代码。

☑ arch/arm/mach-msm/include/mach/：MSM 平台头文件的目录，可以在内核空间中被其他部分引用。

3.2.4 Makefile 文件

Makefile 文件系统移植的核心，MSM 平台中的对应文件是 arch/arm/machlmsm/Makefile，其主要代码如下所示。

```
obj-y += io.o irq.o timer.o dma.o memory.o
obj-$(CONFIG_ARCH_QSD8X50) += sirc.o
obj-y += devices.o pwrtest.o
obj-y += proc_comm.o
obj-y += dex_comm.o
```

```
obj-y += amss_para.o
obj-y += pmic_global.o
obj-y += vreg.o
obj-y += pmic.o
obj-y += remote_spinlock.o
obj-$(CONFIG_ARCH_MSM_ARM11) += acpuclock-arm11.o idle.o
obj-$(CONFIG_ARCH_QSD8X50) += arch-init-scorpion.o acpuclock-scorpion.o
obj-$(CONFIG_ARCH_MSM7X30) += acpuclock-7x30.o internal_power_rail.o
obj-$(CONFIG_ARCH_MSM7X30) += clock-7x30.o arch-init-7x30.o socinfo.o
obj-$(CONFIG_ARCH_MSM7X30) += rpc_pmapp.o smd_rpcrouter_clients.o spm.o
obj-$(CONFIG_ARCH_MSM7X30) += rpc_hsusb.o
obj-$(CONFIG_ARCH_MSM_SCORPION) += idle-v7.o
obj-y += gpio.o generic_gpio.o
obj-y += nand_partitions.o
obj-y += drv_callback.o
obj-$(CONFIG_ARCH_QSD8X50) += pmic.o htc_wifi_nvs.o htc_bluetooth.o
obj-$(CONFIG_MSM_FIQ_SUPPORT) += fiq_glue.o
obj-$(CONFIG_MACH_TROUT) += board-trout-rfkill.o
obj-$(CONFIG_MSM_SMD) += smd.o smd_debug.o
obj-$(CONFIG_MSM_SMD) += smd_tty.o smd_qmi.o
obj-$(CONFIG_MSM_SMD) += smem_log.o
obj-$(CONFIG_MSM_SMD) += last_radio_log.o
obj-$(CONFIG_MSM_SMD) += htc_port_list.o
ifndef CONFIG_ARCH_MSM7X30
obj-$(CONFIG_MSM_ONCRPCROUTER) += smd_rpcrouter.o
else
obj-$(CONFIG_MSM_ONCRPCROUTER) += smd_rpcrouter-7x30.o
endif
obj-$(CONFIG_MSM_ONCRPCROUTER) += smd_rpcrouter_device.o
ifndef CONFIG_ARCH_MSM7X30
obj-$(CONFIG_MSM_ONCRPCROUTER) += smd_rpcrouter_servers.o
else
obj-$(CONFIG_MSM_ONCRPCROUTER) += smd_rpcrouter_servers-7x30.o
endif
obj-$(CONFIG_MSM_ONCRPCROUTER) += smd_rpcrouter_xdr.o
obj-$(CONFIG_MSM_RPCSERVERS) += rpc_server_dog_keepalive.o
obj-$(CONFIG_MSM_RPCSERVERS) += rpc_server_time_remote.o
obj-$(CONFIG_MSM_DALRPC) += dal.o
obj-$(CONFIG_MSM_DALRPC_TEST) += dal_remotetest.o
obj-$(CONFIG_ARCH_MSM7X30) += dal_axi.o
obj-$(CONFIG_MSM_ADSP) += qdsp5/
obj-$(CONFIG_MSM_ADSP_COMP) += qdsp5_comp/
obj-$(CONFIG_MSM7KV2_AUDIO) += qdsp5v2/
obj-$(CONFIG_QSD_AUDIO) += qdsp6/
obj-$(CONFIG_MSM_HW3D) += hw3d.o
obj-$(CONFIG_PM) += pm.o
obj-$(CONFIG_CPU_FREQ) += cpufreq.o

obj-$(CONFIG_HTC_ACOUSTIC) += htc_acoustic.o
obj-$(CONFIG_HTC_ACOUSTIC_QSD) += htc_acoustic_qsd.o
```

```
obj-$(CONFIG_MSM7KV2_AUDIO) += htc_acoustic_7x30.o
obj-$(CONFIG_SENSORS_AKM8976) += htc_akm_cal.o
obj-$(CONFIG_MACH_HALIBUT) += board-halibut.o board-halibut-panel.o
obj-$(CONFIG_MACH_HALIBUT) += board-halibut-keypad.o fish_battery.o clock.o
obj-$(CONFIG_MACH_SWORDFISH) += board-swordfish.o clock.o
obj-$(CONFIG_MACH_SWORDFISH) += board-swordfish-keypad.o fish_battery.o
obj-$(CONFIG_MACH_SWORDFISH) += board-swordfish-panel.o
obj-$(CONFIG_MACH_SWORDFISH) += board-swordfish-mmc.o
obj-$(CONFIG_MACH_TROUT) += board-trout.o board-trout-gpio.o clock.o
obj-$(CONFIG_MACH_TROUT) += board-trout-keypad.o board-trout-panel.o
obj-$(CONFIG_MACH_TROUT) += htc_akm_cal.o htc_wifi_nvs.o htc_acoustic.o
obj-$(CONFIG_MACH_TROUT) += board-trout-mmc.o board-trout-wifi.o
obj-$(CONFIG_MACH_TROUT) += devices_htc.o
obj-$(CONFIG_MACH_MAHIMAHI) += board-mahimahi.o board-mahimahi-panel.o clock.o
obj-$(CONFIG_MACH_MAHIMAHI) += board-mahimahi-keypad.o board-mahimahi-mmc.o
obj-$(CONFIG_MACH_MAHIMAHI) += board-mahimahi-rfkill.o htc_wifi_nvs.o
obj-$(CONFIG_MACH_MAHIMAHI) += board-mahimahi-wifi.o board-mahimahi-audio.o
obj-$(CONFIG_MACH_MAHIMAHI) += msm_vibrator.o
obj-$(CONFIG_MACH_MAHIMAHI) += board-mahimahi-microp.o
obj-$(CONFIG_MACH_MAHIMAHI) += htc_acoustic_qsd.o
obj-$(CONFIG_MACH_MAHIMAHI) += board-mahimahi-flashlight.o
```

因为 MSM 处理器既有 ARM11 体系结构（属于 ARMv6）的 MSM7k，也有 SCORPION 体系结构（属于 ARMv7）的 QSD8k 等多个系列，所以其不同的方面在 Makefile 中对此做出了区分。在为 mahimahi 板构建的系统中，CONFIG ARCH_MSM_SCORPION、CONFIG_MSM_QDSP6、CONFIG MACH_ SWORDFISH 和 CONFIG- MACH_MAHIMAHI 等几个宏均为真。

3.2.5 驱动和组件

在 MSM 平台中，几乎没有专门针对 Android 的驱动和组件，这是因为几乎都和硬件无关。唯一的区别是，在配置文件中对 Android 专用驱动和组件的选择不同。文件 MSM_defconfig 的主要代码如下所示。

```
# Power management options
CONFIG_PM=y
# CONFIG_PM_DEBUG is not set
CONFIG_PM_SLEEP=y
CONFIG_SUSPEND=y
CONFIG_SUSPEND_FREEZER=y
CONFIG_HAS_WAKELOCK=y
CONFIG_HAS_EARLYSUSPEND=y
CONFIG_WAKELOCK=y
CONFIG_WAKELOCK_STAT=y
CONFIG_USER_WAKELOCK=y
CONFIG_EARLYSUSPEND=y
# CONFIG_NO_USER_SPACE_SCREEN_ACCESS_CONTROL is not set
# CONFIG_CONSOLE_EARLYSUSPEND is not set
CONFIG_FB_EARLYSUSPEND=y
# CONFIG_APM_EMULATION is not set
# CONFIG_PM_RUNTIME is not set
CONFIG_ARCH_SUSPEND_POSSIBLE=y
CONFIG_NET=y
```

```
# Android
CONFIG_ANDROID=y
CONFIG_ANDROID_BINDER_IPC=y
CONFIG_ANDROID_LOGGER=y
CONFIG_ANDROID_RAM_CONSOLE=y
CONFIG_ANDROID_RAM_CONSOLE_ENABLE_VERBOSE=y
CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION=y
CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION_DATA_SIZE=128
CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION_ECC_SIZE=16
CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION_SYMBOL_SIZE=8
CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION_POLYNOMIAL=0x11d
# CONFIG_ANDROID_RAM_CONSOLE_EARLY_INIT is not set
CONFIG_ANDROID_TIMED_OUTPUT=y
CONFIG_ANDROID_TIMED_GPIO=y
CONFIG_ANDROID_LOW_MEMORY_KILLER=y


# RCU Subsystem
# RTC interfaces
# CONFIG_RTC_INTF_SYSFS is not set
# CONFIG_RTC_INTF_PROC is not set
# CONFIG_RTC_INTF_DEV is not set
CONFIG_RTC_INTF_ALARM=y
CONFIG_RTC_INTF_ALARM_DEV=y
# CONFIG_RTC_DRV_TEST is not set
# CONFIG_RTC_DRV_DS1307 is not set
# CONFIG_RTC_DRV_DS1374 is not set
# CONFIG_RTC_DRV_DS1672 is not set
# CONFIG_RTC_DRV_MAX6900 is not set
# CONFIG_RTC_DRV_RS5C372 is not set
# CONFIG_RTC_DRV_ISL1208 is not set
# CONFIG_RTC_DRV_X1205 is not set
# CONFIG_RTC_DRV_PCF8563 is not set
# CONFIG_RTC_DRV_PCF8583 is not set
# CONFIG_RTC_DRV_M41T80 is not set
# CONFIG_RTC_DRV_S35390A is not set
# CONFIG_RTC_DRV_FM3130 is not set
# CONFIG_RTC_DRV_RX8581 is not set
# CONFIG_RTC_DRV_RX8025 is not set

#
# Generic Driver Options
CONFIG_UEVENT_HELPER_PATH=""
# CONFIG_DEVTMPFS is not set
CONFIG_STANDALONE=y
CONFIG_PREVENT_FIRMWARE_BUILD=y
CONFIG_FW_LOADER=y
# CONFIG_FIRMWARE_IN_KERNEL is not set
CONFIG_EXTRA_FIRMWARE=""
# CONFIG_DEBUG_DRIVER is not set
```

```
# CONFIG_DEBUG_DEVRES is not set
# CONFIG_SYS_HYPERVISOR is not set
# CONFIG_CONNECTOR is not set
CONFIG_MTD=y
# CONFIG_MTD_DEBUG is not set
# CONFIG_MTD_TESTS is not set
# CONFIG_MTD_CONCAT is not set
CONFIG_MTD_PARTITIONS=y
# CONFIG_MTD_REDBOOT_PARTS is not set

CONFIG_MTD_CMDLINE_PARTS=y
# CONFIG_MTD_AFS_PARTS is not set
# CONFIG_MTD_AR7_PARTS is not set

CONFIG_ANDROID_PMEM=y

CONFIG_ANDROID_PARANOID_NETWORK=y
```

3.2.6 设备驱动

（1）显示驱动

MSM 平台中的显示驱动是 framebuffer，另外还调用了一些内部独有的功能。在 MSM 平台中，有如下两个和显示功能相关的头文件。

☑ 文件 arch/arm/mach-msm/include/mach/msm_fb.h：这是 framebuffer 驱动程序的头文件。

☑ 文件 include/linux/msm_mdp.h：显示模块头文件。

在 drivers/video/目录中，除了有关 framebuffer 驱动程序的通用代码之外，MSM 显示部分的驱动程序主要被保存在 drivers/video/msm/目录中，例如 gpu 目录包含了图形处理单元（Graphic Process Unit）部分相关的内容。

文件 msm_fb.c 是 framebuffer 驱动程序的入口文件，另外有一些和 mddi（Display Digital Interface，一种串行总线，用于连接 LCD）、mdp（Display Processor，显示的主模块，为 framebuffer 核心使用）实现相关的文件。

文件 msm_mddi（mddi.c）、msm_mdp（mdp.c）和 msm_panel（msm fb.c）等几个 platform_driver 都是和显示部分相关的。文件 msm2/arch/arm/mach-msm/device.c 中定义了对应 msm_mddi 和 msm_mdp 的 platform_device，mddi_client-XXX 中定义了对应 msm_panel 的 platform_device。这 3 个平台驱动可以在 sys 文件系统的目录/sys/bus/platform/drivers/中找到。

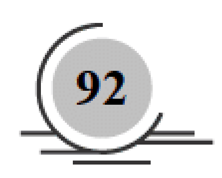
MDP 还定义了一种名为 msm_mdp 的 class。在 sys 文件系统的/sys/class/中保存了其相关信息。

（2）触摸屏驱动

MSM 的 mahimahip 平台触摸屏的驱动程序保存在 drivers/input/touchscreen 目录中的文件 synaptics_i2c-rmi.c 和 msm_ts.c 中，它们分别是一个 event 设备。

文件 synaptics_i2c_rmi.c 中的驱动是一个 i2c 的触摸屏的驱动程序，其 i2c_driver 的名称为 synaptics-rmi-ts。在文件 arch/arm/mach-msm/board-mahimahi.c 中定义其对应的 i2c_device，这个驱动在 sys 文件系统的 sys/bu s/i2c/drivers/synaptics-rmi-ts 目录中，它在 i2c-0 总线上的 id 为 0040。

文件 synaptlcs_i2c_rmi.c 对应的 event 设备是/dev/inpuUevent2。文件 msmts.c 是高通 MSM/QSD 触摸屏的驱动程序，在 sys 文件系统的目录/sys/bus/platform/drivers/中可以找到其相关的信息，文件 msm2/arch/arm/



mach-msm/device.c 定义了相对应的 platform_device。

（3）按键和轨迹球驱动

MSM 的 mahimahip 平台系统包含了按键（有 3 个按键）和轨迹球的功能，具体功能是通过文件 arch/arm/mach-msm/board-mahimahi-keypad.h 实现的，在此文件中注册了名为 mahimahi-keypad 的键盘设备和名称为 mahimahi-nav 的轨迹球设备，对应的设备节点分别为/dev/input/event4 和/dev/inpuUevent5。

（4）时钟驱动

MSM 的实时时钟的驱动程序在 drivers/rtc 的 rtc-MSM7kOOa.c 和 hctosys.c 文件中实现。文件 rtc-MSM7kOOa.c 实现了标准的实时时钟的实现。驱动程序名称为 rs30000048:00010000，sys 文件系统中可以在/sys/bus/platform/drivers/中找到。

文件 hctosys.c 中提供了实时时钟的初始化函数。

（5）摄像头驱动

MSM 的摄像头系统构成的方式为经典的 Camera 驱动+Sensor 驱动方式。其驱动程序是基于 Video for Linux2 的摄像头驱动程序。

除了 v412 的共用部分以外，MSM 的主要文件在 drivers/media/video/msm/目录中，里面包含了 msm_v412.c、msm_camera.c、s5k3e2fx.c、msm_vfe8x_proc.c 等文件。

文件 msm_camera.c 是公用的库函数，创建了/dev/msm_camera 中的各个设备文件。在此主要包含了 3 个自定义的字符设备，其中，frame0 为帧数据设备，config0 为配置设备，control0 为控制设备。

文件 include/media/msm_camera.h 是 MSM 摄像头相关的头文件，其中定义了各种额外的 ioctl 命令。

文件 msm_v412.c 是 v412 驱动程序的实现文件，实现了标准的 Video for Linux 2 的驱动程序，它实际上是在调用 msm_camera.c 中的内容基础上实现的。

s5k3e2fx 是摄像头传感器的驱动程序，platform_driver 的名称为 msm_camera_s5k3e2fx，这个名称和 board-mahimahi.c 中定义的 platform_device 相匹配。

s5k3e2fx 是连接在 i2c 总线上的，其地址为 0-0010，在 sys 文件系统中。

（6）无线局域网驱动

MSM 平台包含了无线局域网，使用 bcm4329。bcm4329 是集成了蓝牙、无线局域网、FM 为一体的芯片，相关代码内容在 drivers/net/wireless/bcm4329 目录中。其中，在文件 dhd_linux.c 中定义了 platform_driver 的名称为 bcm4329_wlan，其名称和 board-mahimahi-wifi.c 中定义的 platform_device 相匹配。

（7）蓝牙驱动

MSM 的 mahimahip 平台的蓝牙驱动使用标准的 HCI 驱动，路径在 drivers/bluetooth 中，包括 hci 11.c、hci_h3.c 和 hci_ldisc.c，编译后将生成 hci uart.o 文件。

（8）DSP 驱动

在 MSM 平台的 DSP（数字信号处理器）用于实现比较高级的功能，主要被保存在如下的目录中。

☑ arch/arm/mach-msm/qdsp5：MSM7k 系列处理器使用的 5 代 DSP。

☑ arch/arm/mach-msm/qdsp6：QSD8k 系列处理器使用的 6 代 DSP。

其中，在 arch/arm/mach-msm/qdsp6 目录中包含如下文件。

☑ dal.c：dal 协议文件。

☑ q6audio.c：Audio 系统通用库文件。

☑ audio_ctl.c：音频控制文件。

☑ routing.c：音频路径控制。

☑ pcm_in.c：PCM 输入通道。

☑ pcm_out.c：PCM 输出通道。

☑ mp3.c：MP3 码流直接输出通道。

☑ msm_q6vdec.c：视频解码。

☑ msm_q6venc.c：视频编码。

Audio 系统的头文件是 arch/arm/mach-ms m/include/mach/msm_qdsp6_audio.h。MSM 视频编解码的头文件被保存在 include/linux/目录中，主要由如下两个文件实现。

☑ msm_q6vdec.h：视频解码器头文件。

☑ msm_q6venc.h：视频编码器头文件。

q6venc 是视频编码器在用户空间的节点，是一个 MISC 字符设备，vdec 是视频解码器在用户空间的节点，是一个自定义的字符设备。

3.2.7 高通特有的组件

在 MSM 处理器中还包含了很多高通独有的组件驱动，这些驱动的实现文件被保存在 arch/arm/mach-msm/目录中，主要内容如下。

☑ smd_private.h：共享内存相关的结构和内存区域等定义。

☑ smd.c：共享内存的部分底层机制的实现。

☑ proc_comm.c：处理器间简单远程命令接口实现。

☑ smd_rpcrouter.c：ONCRPC 实现部分。

☑ smd_rpcrouter_device.c：ONCRPC 实现部分。

☑ smd_rpcrouter_servers.c：ONCRPC 实现部分。

（1）SMEM

SMEM（Shared Memory）用于管理共享内存的区域，有静态和动态两种区域。静态区域一般是定义好的，可以由两个 CPU 分别直接访问，而动态区域一般通过 SMEM 的分配机制来分配。

SMEM 是最基础的共享内存管理机制，所有使用共享内存的通信机制或协议都基于它来实现。区域很多，有用于存放基本的版本等信息的，也有用于实现简单的 RPC 机制的，还有分配 Buffer 以用于大量数据传输的。

SMEM 的区域定义在 arch/arm/mach-msm/目录 smd_private.h 中，实现代码大多在该目录下的 smd.c 文件中。

（2）SMSM

SMSM 利用 SMEM 中的 SMEM_SMSM_SHARED_STATE 等区域，传送两个 CPU 的状态信息，如 modem 重启、休眠等状态。

当 SMSM 信息变化后，通常通过中断来通知到另一处理器。

（3）PROC COMM

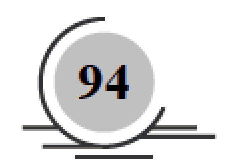
PROC COMM 使用 SMEM 中的最前面一个区域：SMEM_PROC_COMM。它是一套应用处理器向 MODEM 发送简单命令的接口。

PROC COMM 能传递的信息非常有限，仅能传递两个 uint32 的数据作为参数，也只能接收两个 uint 32 的数据，加一个 boolean 作为返回值。但相对于后面提到的 RPC，PROC COMM 更轻量级。

PROC COMM 定义在文件 proc_comm.c 中，通常应用处理器会使用 msm_proc_comm 接口函数来发送命令，并通过轮询进行等待返回。注意需要支持的命令，要在 modem 侧启动时，注册好对应的处理程序。

常用的 PROC COMM 命令如下。

☑ SMEM_PROC_COMM_GET_BAIT_LEVEL：获取电池电量级别。



- ☑ SMEM_PROC_COMM_CHG_IS_CHARGING：判断是否在充电。
- ☑ SMEM_PROC_COMM_POWER_ DOWN：关机。
- ☑ SMEM_PROC_COMM_RESET_MODEM：重启 modem。

（4）SMD

SMD 用于处理器之间，是一套通过共享内存同步大量数据的协议。目前 SMD 支持 64 个通道，其中 36 个已经定义。分别用于蓝牙、RPC、modem 数据链接等。为了防止冲突，每个通道使用两路连接，将发送和接收分开。

SMD 使用 SMEM 中的对应区域分配适当大小的缓冲，并定义了详细的协议，用于控制传输的开启、停止等。控制的标记类似于 RS-232，而且支持流控。

SMD 支持 stream 模式和 packet 模式。后者会对数据进行封包，保证对端获取到的数据与传送时分块一致。

SMD 主要是在文件 smd.c 中实现的，在里面有一整套如下函数的接口。

- ☑ smd_open：打开一个 smd 通道。
- ☑ smd_close：关闭一个 smd 通道。
- ☑ smd_read：从一个通道中读取。
- ☑ smd_write：写入到一个通道。
- ☑ smd_alloc_channel：分配一个通道。

（5）ONCRPC

RPC 的含义为 Remote Procedure Calls（远程过程调用）。此处特指处理器间的远程过程调用。在高通平台中，这一机制又叫 ONCRPC（Open Network Computing Remote Procedure Call），以下提及的 ONCRPC，都是特指高通平台上的具体实现。

ONCRPC 基于共享内存上的 SMD 实现。使应用处理器端的应用程序可以直接访问 modem 端的服务，支持的服务如下。

- ☑ Call Manager（CM API）。
- ☑ Wueless Messaging Service（WMS API）。
- ☑ GSDI（Sm，USIM）。
- ☑ GSTK（Toolkit）。
- ☑ PDSM API（GPS）。

另外，ONCRPC 基于服务端/客户端的思想构建，在以 smd_rpcrouter 开头的源文件中实现。通过服务端实现访问 modem 服务的工作，而客户端负责将公开的 API 交付给用户程序调用。用户程序如果需要使用 ONCRPC，需要链接 ONCRPC-shared 和 AMSS RPC exported 等库。

第 2 篇



Android 专有驱动篇

- 第 4 章　分析硬件抽象层
- 第 5 章　Binder 通信驱动详解
- 第 6 章　Logger 驱动架构详解
- 第 7 章　Ashmem 驱动详解
- 第 8 章　搭建测试环境
- 第 9 章　低内存管理驱动

第 4 章　分析硬件抽象层

在 Android 系统中，硬件抽象层（Hardware Abstract Layer，HAL）在用户空间中运行。HAL 能够向下屏蔽硬件驱动模块的实现细节，向上提供硬件访问服务。通过硬件抽象层，Android 系统通过如下两层来支持硬件设备。

☑　第一层在用户空间中实现。

☑　第二层在内核空间中实现。

本章将详细讲解 Android 5.0 中 HAL 源码的基本知识，为读者学习本书后面高级知识打下基础。

4.1　HAL 基础

HAL 层（硬件抽象层）是位于操作系统内核与硬件电路之间的接口层，其目的在于将硬件抽象化。它隐藏了特定平台的硬件接口细节，为操作系统提供虚拟硬件平台，使其具有硬件无关性，这样就可以在多种平台上进行移植。从软硬件测试的角度来看，软硬件的测试工作都可分别基于硬件抽象层来完成，从此使软硬件测试工作的并行进行成为可能。本节将简要介绍 HAL 的基础知识。

4.1.1　推出 HAL 的背景

在 Android 系统中，推出 HAL 的目的是为了保护一些硬件提供商的知识产权，为了避开 Linux 的 GPL 束缚。Google 架构师的思路是把控制硬件的动作都放到了 Android HAL 中，而 Linux Driver（驱动）仅负责完成一些简单的数据交互作用，甚至把硬件寄存器空间直接映射到 User Space（用户空间）。而 Android 系统是基于 Aparch 的 License，因此硬件厂商可以只提供二进制代码，所以说 Android 只是一个开放的平台，并不是一个开源的平台。也许正是因为 Android 不遵从 GPL，所以 Greg Kroah-Hartman 才在 2.6.33 内核将 Android 驱动从 Linux 中删除，当然从后来 Linux 3.3 版本开始又将 Android 重新纳入进来，GPL 和硬件厂商目前还是有着无法弥合的裂痕。

Android 系统为什么要把对硬件的支持划分为两层来实现呢？具体来说有如下两个原因。

（1）Linux 内核源代码是遵循 GPL1 协议的，即如果在 Android 系统所使用的 Linux 内核中添加或者修改了代码，那么就必须将它们公开。因此，如果 Android 系统像其他的 Linux 系统一样，把对硬件的支持完全实现在硬件驱动模块中，那么就必须将这些硬件驱动模块源代码公开，这样就可能会损害移动设备厂商的利益，因为这相当于暴露了硬件的实现细节和参数。

（2）Android 系统源代码是遵循 Apache License2 协议的，它允许移动设备厂商添加或者修改 Android 系统源代码，而又不必公开这些代码。因此，如果把对硬件的支持完全实现在 Android 系统的用户空间中，那么就可以隐藏硬件的实现细节和参数。然而，这是无法做到的，因为只有内核空间才有特权操作硬件设备。一个折中的解决方案便是将对硬件的支持分别实现在内核空间和用户空间中，其中，内核空间仍然是以硬件驱动模块的形式来支持，不过它只提供简单的硬件访问通道；而用户空间以硬件抽象层模块的形式来支持，它封装了硬件的实现细节和参数，这样就可以保护移动设备厂商的利益了。

在 Android 系统中可以分为如下 6 种 HAL。

☑　上层软件。
☑　内部以太网。
☑　内部通信 CLIENT。
☑　用户接入口。
☑　虚拟驱动，设置管理模块。
☑　内部通信 SERVER。

在 Android 系统中，定义硬件抽象层接口的代码具有以下 5 个特点。

☑　硬件抽象层具有与硬件密切的相关性。
☑　硬件抽象层具有与操作系统无关性。
☑　接口定义的功能应包含硬件或系统所需硬件支持的所有功能。
☑　接口定义简单明了，太多接口函数会增加软件模拟的复杂性。
☑　具有可测性的接口设计有利于系统的软硬件测试和集成。

在 Android 源码中，HAL 主要被保存在下面的目录中。

☑　libhardware_legacy：过去的目录，采取了链接库模块观念来架构。
☑　libhardware：新版的目录，被调整为用 HAL stub 观念来架构。
☑　ril：是 Radio 接口层。
☑　msm7k：和 QUAL 平台相关的信息。

4.1.2　HAL 的基本结构

在 Android 系统中，HAL 的位置结构如图 4-1 所示。



图 4-1　HAL 层结构

从图 4-1 所示的结构图可以看出，HAL 的功能是把 Android Framework（Android 框架）与 Linux 内核（Linux 内核）隔离。这样 Android 可以不过度依赖 Linux Kernel，从而在不考虑驱动程序的前提下进行 Framework 层的应用开发工作。HAL 层主要包含了 GPS、Vibrator、WiFi、Copybit、Audio、Camera、Lights、Ril、Overlay 等模块。

目前 Android 的 HAL 层仍然分布在不同的位置，所以如 Camera、WiFi 等目录并不包含所有的 HAL 程序代码。在 HAL 架构成熟前的结构如图 4-2 所示，现在 HAL 层的结构如图 4-3 所示。

从现在 HAL 层的结构可以看出，当前的 HAL Stub 模式是一种代理人（proxy）的概念，虽然 Stub 仍以 *.so 档的形式存在，但是 HAL 已经将*.so 档隐藏了。Stub 向 HAL 提供了功能强大的操作函数（Operations），而 runtime 则从 HAL 获取特定模块（stub）的函数，然后再回调这些操作函数。这种以 Indirect Function Call

模式的架构，让 HAL stub 变成了一种“包含”关系，即在 HAL 中包含了许多 stub（代理人）。Runtime 只要说明 module ID（类型）就可以取得操作函数。在当前的 HAL 模式中，Android 定义了 HAL 层结构框架，这样通过接口访问硬件时就形成了统一的调用方式。

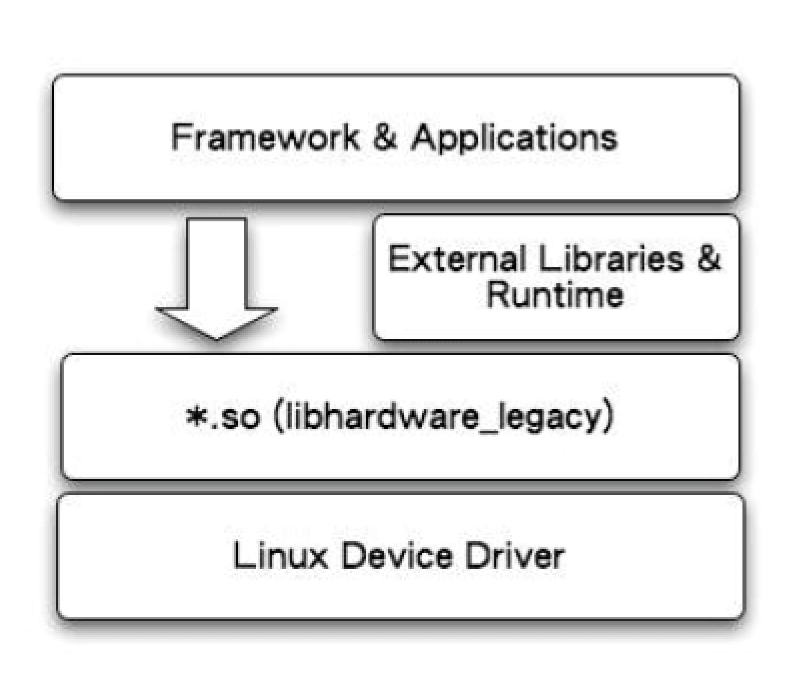


图 4-2　成熟前的 HAL 架构

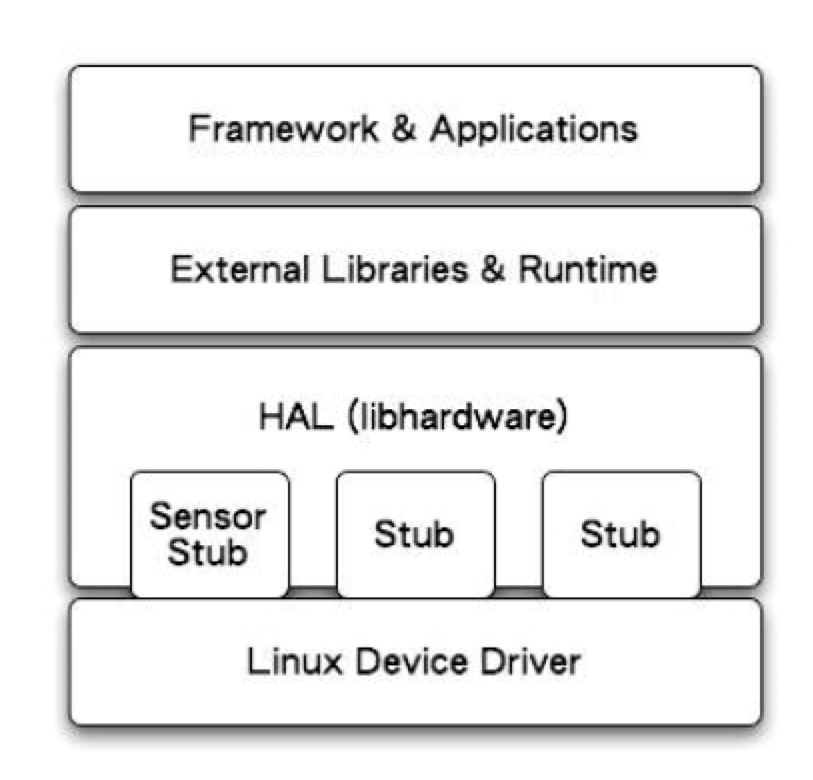


图 4-3　现在的 HAL 架构

注意：HAL_legacy和HAL的对比

为了使读者明白过去结构和现在结构的差别，接下来将对HAL_legacy和HAL做一个对比。

（1）HAL_legacy

这是过去HAL的模块，采用共享库形式，在编译时会调用到。由于采用function call形式来调用，因此可被多个进程使用，但会被mapping到多个进程空间中造成浪费，同时需要考虑代码能否安全重入的问题（thread safe）。

（2）HAL

这是新式的HAL，采用了HAL module和HAL stub结合形式。HAL stub不是一个共享库，在编译时上层只拥有访问HAL stub的函数指针，并不需要HAL stub。在上层通过HAL module提供的统一接口获取并操作HAL stub，所以文件只会被映射到一个进程，而不会存在重复映射和重入问题。

在 Android 系统中，HAL 层的源码结构如下。

（1）/hardware/libhardware_legacy/：旧的 HAL 架构，采取链接库模块的方式。

（2）/hardware/libhardware：新的 HAL 架构，调整为 HAL stub，具体目录结构如下。

- ☑ /hardware/libhardware/hardware.c：编译成 libhardware.s，置于/system/lib。
- ☑ /hardware/libhardware/include/hardware 目录下包含如下头文件。
  - ➢ hardware.h：通用硬件模块头文件。
  - ➢ copybit.h：copybit 模块头文件。
  - ➢ gralloc.h：gralloc 模块头文件。
  - ➢ lights.h：背光模块头文件。
  - ➢ overlay.h：overlay 模块头文件。
  - ➢ qemud.h：qemud 模块头文件。
  - ➢ sensors.h：传感器模块头文件。
- ☑ /hardware/libhardware/modules：在此目录下定义了很多硬件模块，例如/hardware/msm7k、/hardware/qcom、/hardware/ti、/device/Samsung、/device/moto，这些是各个厂商平台相关的 HAL。

4.2　分析 HAL module 架构

Android 5.0 的 HAL 采用 HAL module 和 HAL stub 结合的形式进行架构，HAL stub 不是一个 Share Library（共享程序），在编译时上层只拥有访问 HAL stub 的函数指针，并不需要 HAL stub。上层通过 HAL module 提供的统一接口获取并操作 HAL stub，so 文件只会被 mapping 到一个进程，也不存在重复 mapping 和重入问题。

在 Android 5.0 系统中，HAL module 架构主要分为如下 3 个结构体。

- ☑ struct hw_module_t
- ☑ struct hw_module_methods_t
- ☑ struct hw_device_t

上述 3 个结构的继承关系如图 4-4 所示。

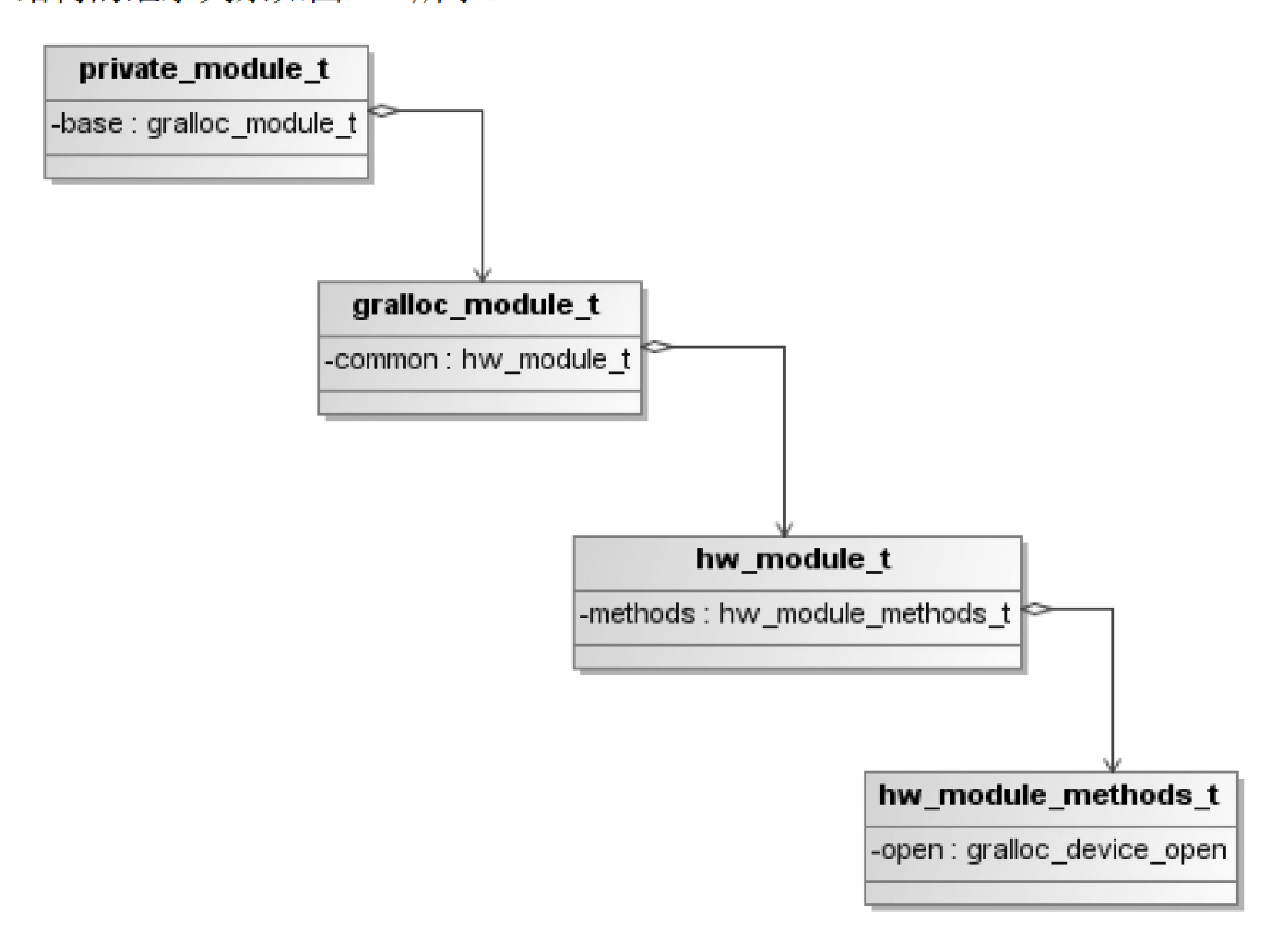


图 4-4　Android HAL 结构的继承关系

以上 3 个抽象概念在文件 hardware.c 中进行了具体描述，而 HAL 模块的源代码保存在 harware 目录中。对于不同的 hardware 的 HAL，对应的 lib 命名规则是 id.variant.so，例如 gralloc.msm7k.so 表示其 id 是 gralloc，msm7k 是 variant。variant 的取值范围是在该文件中定义的 variant_keys 对应的值。

4.2.1　结构体 hw_module_t

结构 hw_module_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下所示。

```
typedef struct hw_module_t {
    uint32_t tag;
```

```
    uint16_t module_api_version;
#define version_major module_api_version
    uint16_t hal_api_version;
#define version_minor hal_api_version
    const char *id;
    const char *name;
    const char *author;
    struct hw_module_methods_t* methods;
    void* dso;
    uint32_t reserved[32-7];

} hw_module_t;
```

在结构体 hw_module_t 中，读者需要注意如下 5 点。

（1）在结构体 hw_module_t 的定义前面有一段注释，意思是，硬件抽象层中的每一个模块都必须自定义一个硬件抽象层模块结构体，而且它的第一个成员变量的类型必须为 hw_module_t。

（2）硬件抽象层中的每一个模块都必须存在一个导出符号 HAL_MODULE_IFNO_SYM，即 HMI，它指向一个自定义的硬件抽象层模块结构体。后面在分析硬件抽象层模块的加载过程时，将会看到这个导出符号的意义。

（3）结构体 hw_module_t 的成员变量 tag 的值必须设置为 HARDWARE_MODULE_TAG，即设置为一个常量值（'H' <<24 | 'W'<<16 | 'M'<<8 | 'T'），用来标志这是一个硬件抽象层模块结构体。

（4）结构体 hw_module_t 的成员变量 dso 用来保存加载硬件抽象层模块后得到的句柄值。前面提到，每一个硬件抽象层模块都对应有一个动态链接库文件。加载硬件抽象层模块的过程实际上就是调用 dlopen 函数来加载与其对应的动态链接库文件的过程。在调用 dlclose 函数来卸载这个硬件抽象层模块时，要用到这个句柄值，因此，我们在加载时需要将它保存起来。

（5）结构体 hw_module_t 的成员变量 methods 定义了一个硬件抽象层模块的操作方法列表，它的类型为 hw_module_methods_t，接下来介绍它的定义，其定义代码如下所示。

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t* module, const char* id,
            struct hw_device_t** device);
} hw_module_methods_t;
```

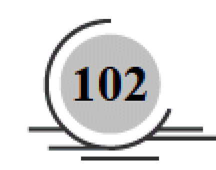
4.2.2 结构体 hw_module_methods_t

结构 hw_module_methods_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下所示。

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t* module, const char* id,
            struct hw_device_t** device);

} hw_module_methods_t;
```

在结构体 hw_module_methods_t 中只有一个成员变量，它是一个函数指针，用来打开硬件抽象层模块中的硬件设备。其中，参数 module 表示要打开的硬件设备所在的模块；参数 id 表示要打开的硬件设备的 ID；参数 device 是一个输出参数，用来描述一个已经打开的硬件设备。由于一个硬件抽象层模块可能会包含多个硬件设备，因此在调用结构体 hw_module_methods_t 的成员变量 open，打开一个硬件设备时需要指定它



的 ID。

4.2.3　结构体 hw_device_t

结构 hw_device_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下所示。

```
typedef struct hw_device_t {
    uint32_t tag;
    uint32_t version;
    struct hw_module_t* module;
    uint32_t reserved[12];
    int (*close)(struct hw_device_t* device);

} hw_device_t;
```

在 Android 系统中，硬件抽象层中的硬件设备使用结构体 hw_device_t 来描述，接下来介绍它的定义，其代码如下所示。

```
typedef struct hw_device_t {
    uint32_t tag;
    uint32_t version;
    struct hw_module_t* module;
    uint32_t reserved[12];
    int (*close)(struct hw_device_t* device);
} hw_device_t;
```

在结构体 hw_device_t 中，需要注意如下 3 点。

（1）硬件抽象层模块中的每一个硬件设备都必须自定义一个硬件设备结构体，而且它的第一个成员变量的类型必须为 hw_device_t。

（2）结构体 hw_device_t 的成员变量 tag 的值必须设置为 HARDWARE_DEVICE_TAG，即设置为一个常量值（'H' <<24 | 'W'<<16 | 'D'<<8 | 'T'），用来标志这是一个硬件抽象层中的硬件设备结构体。

（3）结构体 hw_device_t 的成员变量 close 是一个函数指针，用来关闭一个硬件设备。

4.3　分析文件 hardware.c

文件 hardware.c 是文件 hardware.h 的具体实现，本节将详细分析 Android 5.0 HAL 模块中文件 hardware.c 的基本源码。

4.3.1　寻找动态链接库的地址

函数 hw_get_module()能够根据模块 ID 寻找硬件模块动态链接库的地址，然后调用函数 load 打开动态链接库，并从中获取硬件模块结构体地址。执行后首先是根据固定的符号 HAL_MODULE_INFO_SYM 寻找到结构体 hw_module_t，然后在 hw_moule_t 中 hw_module_methods_t 结构体成员函数提供的结构 open 中打开相应的模块，并同时进行初始化操作。因为用户在调用 open()时通常都会传入一个指向 hw_device_t 指针的指针。这样函数 open()将对模块的操作函数结构保存到结构体 hw_device_t 中，用户通过它可以和模块进行交互。

函数 hw_get_module()的实现代码如下所示。

```
int hw_get_module(const char *id, const struct hw_module_t **module)
{
    int status;
    int i;
    const struct hw_module_t *hmi = NULL;
    char prop[PATH_MAX];
    char path[PATH_MAX];
  /* Loop through the configuration variants looking for a module */
    for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++) {
/*
```

4.3.2 数组 variant_keys

在函数 hw_get_module()中需要用到数组 variant_keys，因为 HAL_VARIANT_KEYS_COUNT 表示数组 variant_keys 的大小。定义数组 variant_keys 的代码如下所示。

```
* static const char *variant_keys[] = {
*     "ro.hardware",   /* This goes first so that it can pick up a different
*                         file on the emulator. */
*     "ro.product.board",
*     "ro.board.platform",
*     "ro.arch"
* };
```

然后通过此数组，并使用如下代码得到操作权限。

```
        if (i < HAL_VARIANT_KEYS_COUNT) {
            if (property_get(variant_keys[i], prop, NULL) == 0) {
                continue;
            }
```

此处的 variant_keys[i]对应有 3 个值，分别是 trout、msm7k 和 ARMv6。

接下来通过如下代码将路径和文件名保存到 path。

```
  snprintf(path, sizeof(path), "%s/%s.%s.so",
    HAL_LIBRARY_PATH, id, prop);
```

通过上述代码，把 HAL_LIBRARY_PATH/id.***.so 保存到 path 中，其中***就是上面 variant_keys 中各个元素所对应的值。

4.3.3 载入相应的库

载入相应的库，并把它们的 HMI 保存到 module 中，具体代码如下所示。

```
        } else {
            snprintf(path, sizeof(path), "%s/%s.default.so",
                    HAL_LIBRARY_PATH, id);
        }
        if (access(path, R_OK)) {
            continue;
        }
        /* we found a library matching this id/variant */
        break;
    }
```

```
    status = -ENOENT;
    if (i < HAL_VARIANT_KEYS_COUNT+1) {
        /* load the module, if this fails, we're doomed, and we should not try
         * to load a different variant. */
        status = load(id, path, module);//load 相应库，并把它们的 HMI 保存到 module 中
    }
  return status;
}
```

4.3.4　获得 hw_module_t 结构体

通过函数 load()打开相应的库并获得 hw_module_t 结构体，具体实现代码如下所示。

```
static int load(const char *id,
        const char *path,
        const struct hw_module_t **pHmi)
{
    int status;
    void *handle;
    struct hw_module_t *hmi;
  handle = dlopen(path, RTLD_NOW);//打开相应的库
    if (handle == NULL) {
        char const *err_str = dlerror();
        LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
        status = -EINVAL;
        goto done;
    }

    const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
    hmi = (struct hw_module_t *)dlsym(handle, sym);//获得 hw_module_t 结构体
    if (hmi == NULL) {
        LOGE("load: couldn't find symbol %s", sym);
        status = -EINVAL;
        goto done;
    }

    /* Check that the id matches */
    if (strcmp(id, hmi->id) != 0) {//只是一个 check
        LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
        status = -EINVAL;
        goto done;
    }

    hmi->dso = handle;

    /* success */
    status = 0;
done:
    if (status != 0) {
        hmi = NULL;
        if (handle != NULL) {
```

```
                dlclose(handle);
                handle = NULL;
            }
        } else {
            LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
                    id, path, *pHmi, handle);
        }

        *pHmi = hmi;//得到 hw_module_t

        return status;
}
```

4.4　分析硬件抽象层的加载过程

每一个硬件抽象层模块在内核中都对应有一个驱动程序，硬件抽象层模块就是通过这些驱动程序来访问硬件设备的，它们通过读写设备文件来进行通信。硬件抽象层中的模块接口源文件一般保存在 hardware/libhardware 目录中，其目录结构如图 4-5 所示。

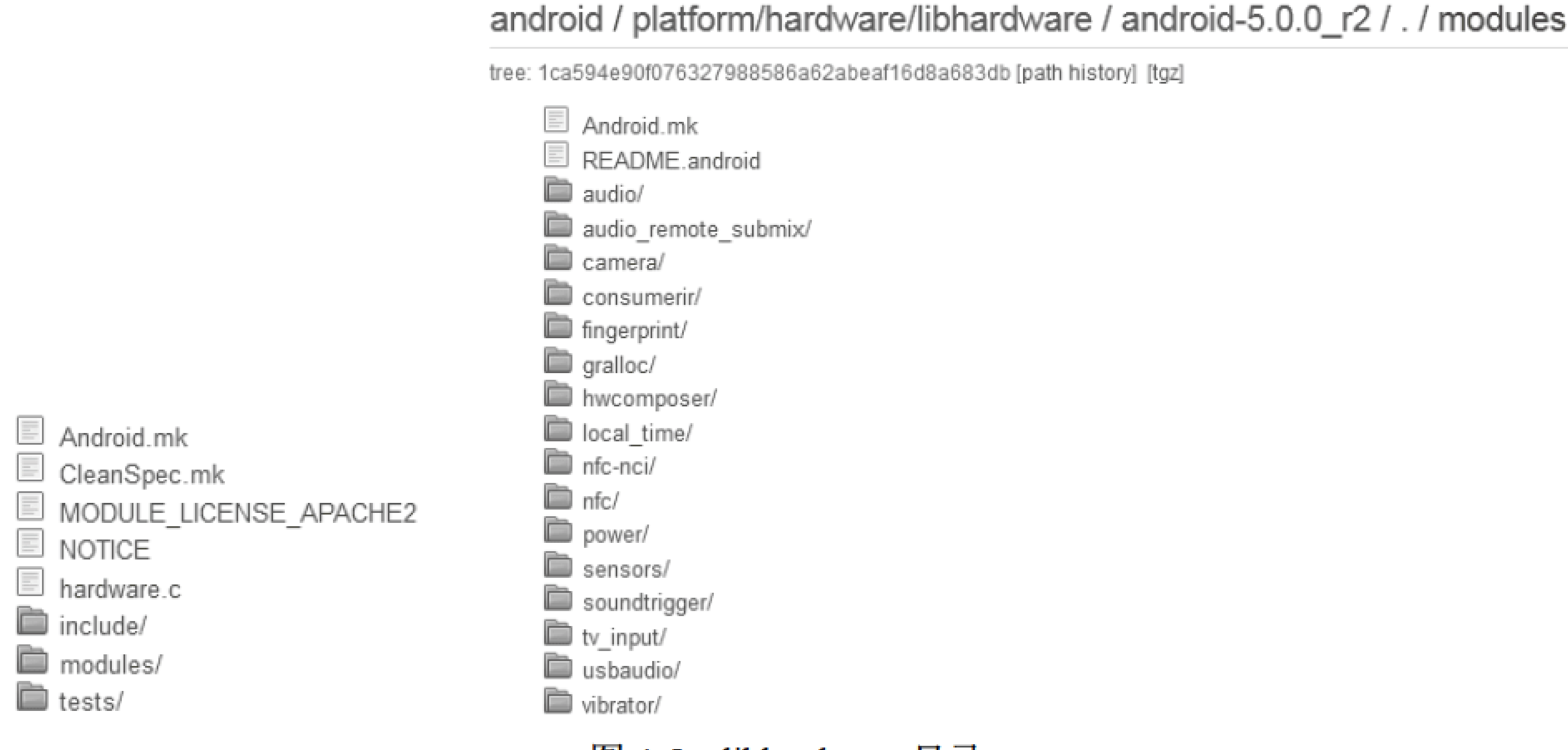


图 4-5　libhardware 目录

Android 系统中的硬件抽象层模块是由系统统一加载的，当调用者需要加载这些模块时，只要指定它们的 ID 值就可以了。在 Android 硬件抽象层中，负责加载硬件抽象层模块的函数是 hw_get_module()，此函数在 hardware/libhardware/include/hardware/hardware.h 文件中定义。

函数 hw_get_module()有 id 和 module 两个参数。其中，id 是输入参数，表示要加载的硬件抽象层模块 ID；module 是输出参数，如果加载成功，那么它指向一个自定义的硬件抽象层模块结构体。函数的返回值是一个整数，如果等于 0，则表示加载成功；如果小于 0，则表示加载失败。函数 hw_get_module()的具体实现代码如下所示。

```
int hw_get_module(const char *id, const struct hw_module_t **module)
{
    return hw_get_module_by_class(id, NULL, module);
}
```

函数 hw_get_module()在文件 hardware/libhardware/hardware.c 中实现，其中数组 variant_keys 用来组装要加载的硬件抽象层模块的文件名称，常量 HAL_VARIANT_KEYS_COUNT 表示数组 variant_keys 的大小，宏 HAL_LIBRARY_PATH1 和 HAL_LIBRARY_PATH2 用来定义要加载的硬件抽象层模块文件所在的目录。第 32 行到第 50 行的 for 循环根据数组 variant_keys 在 HAL_LIBRARY_PATH1 和 HAL_LIBRARY_PATH2 目录中检查对应的硬件抽象层模块文件是否存在，如果存在则结束 for 循环；第 56 行调用 load()函数来执行加载硬件抽象层模块的操作。函数 hw_get_module()的具体实现代码如下所示。

```
16 int hw_get_module(const char *id, const struct hw_module_t **module)
17 {
18 int status;
19 int i;
20 const struct hw_module_t *hmi = NULL;
21 char prop[PATH_MAX];
22 char path[PATH_MAX];
23
24 /*
25 * Here we rely on the fact that calling dlopen multiple times on
26 * the same .so will simply increment a refcount (and not load
27 * a new copy of the library).
28 * We also assume that dlopen() is thread-safe.
29 */
30
31 /* Loop through the configuration variants looking for a module */
32 for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++) {
33 if (i < HAL_VARIANT_KEYS_COUNT) {
34 if (property_get(variant_keys[i], prop, NULL) == 0) {
35 continue;
36 }
37
38 snprintf(path, sizeof(path), "%s/%s.%s.so",
39 HAL_LIBRARY_PATH1, id, prop);
40 if (access(path, R_OK) == 0) break;
41
42 snprintf(path, sizeof(path), "%s/%s.%s.so",
43 HAL_LIBRARY_PATH2, id, prop);
44 if (access(path, R_OK) == 0) break;
45 } else {
46 snprintf(path, sizeof(path), "%s/%s.default.so",
47 HAL_LIBRARY_PATH1, id);
48 if (access(path, R_OK) == 0) break;
49 }
50 }
51
52 status = -ENOENT;
53 if (i < HAL_VARIANT_KEYS_COUNT+1) {
54 /* load the module, if this fails, we're doomed, and we should not try
55 * to load a different variant. */
56 status = load(id, path, module);
57 }
58
```

```
59 return status;
60 }
```

编译好的模块文件位于 out/target/product/generic/system/lib/hw 目录中，而这个目录经过打包后，就对应于设备上的/system/lib/hw 目录。宏 HAL_LIBRARY_PATH2 所定义的目录为/vendor/lib/hw，用来保存设备厂商所提供的硬件抽象层模块接口文件。

在上述第 56 行代码中，调用函数 load()执行硬件抽象层模块的加载操作，此函数的具体实现代码如下所示。

```
static int load(const char *id,
const char *path,
const struct hw_module_t **pHmi)
{
int status;
void *handle;
struct hw_module_t *hmi;

/*
* load the symbols resolving undefined symbols before
* dlopen returns. Since RTLD_GLOBAL is not or'd in with
* RTLD_NOW the external symbols will not be global
*/
handle = dlopen(path, RTLD_NOW);
if (handle == NULL) {
char const *err_str = dlerror();
LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
status = -EINVAL;
goto done;
}

/* Get the address of the struct hal_module_info. */
const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
hmi = (struct hw_module_t *)dlsym(handle, sym);
if (hmi == NULL) {
LOGE("load: couldn't find symbol %s", sym);
status = -EINVAL;
goto done;
}

/* Check that the id matches */
if (strcmp(id, hmi->id) != 0) {
LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
status = -EINVAL;
goto done;
}

hmi->dso = handle;

/* success */
status = 0;

done:
```

```
44  if (status != 0) {
45  hmi = NULL;
46  if (handle != NULL) {
47  dlclose(handle);
48  handle = NULL;
49  }
50  } else {
51  LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
52  id, path, *pHmi, handle);
53  }
54
55  *pHmi = hmi;
56
57  return status;
58  }
```

在上述代码中，第 14 行调用函数 dlopen()将它加载到内存中。加载完成这个动态链接库文件之后，第 24 行就调用函数 dlsym()来获得里面名称为 HAL_MODULE_INFO_SYM_AS_STR 的符号，这个符号指向的是一个自定义的硬件抽象层模块结构体，它包含了对应的硬件抽象层模块的所有信息。HAL_MODULE_INFO_SYM_AS_STR 是一个宏，它的值定义为 HMI。根据硬件抽象层模块的编写规范，每一个硬件抽象层模块都必须包含一个名称为 HMI 的符号，而且这个符号的第一个成员变量的类型必须定义为 hw_module_t，因此，第 24 行可以安全地将模块中的 HMI 符号转换为一个 hw_module_t 结构体指针。获得了这个 hw_module_t 结构体指针之后，第 32 行调用 strcmp()函数来验证加载得到的硬件抽象层模块 ID 是否与所要求加载的硬件抽象层模块 ID 一致。如果不一致，就说明出错了，函数返回一个错误值-EINVAL。最后，第 38 行将成功加载后得到的模块句柄值 handle 保存在 hw_module_t 结构体指针 hmi 的成员变量 dso 中，然后将它返回给调用者。

4.5　分析硬件访问服务

当开发好硬件抽象层模块之后，我们通常还需要在应用程序框架层中实现一个硬件访问服务。硬件访问服务通过硬件抽象层模块来为应用程序提供硬件读写操作。由于硬件抽象层模块是使用 C++语言开发的，而应用程序框架层中的硬件访问服务是使用 Java 语言开发的，因此，硬件访问服务必须通过 Java 本地接口（Java Native Interface，JNI）来调用硬件抽象层模块的接口。本节将详细分析硬件访问服务的基本源码。

4.5.1　定义硬件访问服务接口

Android 系统的硬件访问服务通常运行在系统进程 System5 中，而使用这些硬件访问服务的应用程序运行在另外的进程中，即应用程序需要通过进程间通信机制来访问这些硬件访问服务。Android 系统提供了一种高效的进程间通信机制——Binder 进程间通信机制 6，应用程序就是通过它来访问运行在系统进程 System 中的硬件访问服务的。Binder 进程间通信机制要求提供服务的一方必须实现一个具有跨进程访问能力的服务接口，以便使用服务的一方可以通过这个服务接口来访问它。因此，在实现硬件访问服务之前，我们首先要定义它的服务接口。

在 Android 5.0 系统中，提供了一种描述语言来定义具有跨进程访问能力的服务接口，这种描述语言称为 Android 接口描述语言（Android Interface Definition Language，AIDL）。以 AIDL 定义的服务接口文件是

以 aidl 为后缀名的，在编译时，编译系统会将它们转换成一个 Java 文件，然后再对它们进行编译，本节将使用 AIDL 来定义硬件访问服务接口 IFregService。

在 Android 系统中，通常在 frameworks/base/core/java/android/os 目录中定义硬件访问服务接口，所以把定义了硬件访问服务接口 IFregService 的文件 IFregService.aidl 也保存在这个目录中，其具体代码如下所示。

```
package android.os;
interface IFregService {
void setVal(int val);
int getVal();
}
```

服务接口 IFregService 只定义了两个成员函数，它们分别是 setVal 和 getVal。其中，成员函数 setVal 用来向虚拟硬件设备 freg 的寄存器 val 中写入一个整数，而成员函数 getVal 用来从虚拟硬件设备 freg 的寄存器 val 中读出一个整数。

由于服务接口 IFregService 是使用 AIDL 语言描述的，因此需要将其添加到编译脚本文件中，这样编译系统才能将其转换为 Java 文件，然后再对它进行编译。进入到 frameworks/base 目录中，打开里面的 Android.mk 文件，修改 LOCAL_SRC_FILES 变量的值。

```
LOCAL_SRC_FILES += \
...
voip/java/android/net/sip/ISipService.aidl \
core/java/android/os/IFregService.aidl
```

修改这个编译脚本文件之后，我们就可以使用 mmm 命令对硬件访问服务接口 IFregService 进行编译了。

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/
```

编译后得到的 framework.jar 文件就包含有 IFregService 接口，它继承了 android.os.IInterface 接口。在 IFregService 接口内部，定义了一个 Binder 本地对象类 Stub，它实现了 IFregService 接口，并且继承了 android.os.Binder 类。此外，在 IFregService.Stub 类内部，还定义了一个 Binder 代理对象类 Proxy，同样也实现了 IFregService 接口。

用 AIDL 定义的服务接口是用来进行进程间通信的，其中，提供服务的进程称为 Server 进程，而使用服务的进程称为 Client 进程。在 Server 进程中，每一个服务都对应有一个 Binder 本地对象，它通过一个桩（Stub）来等待 Client 进程发送进程间通信请求。Client 进程在访问运行 Server 进程中的服务之前，首先要获得它的一个 Binder 代理对象接口（Proxy），然后通过这个 Binder 代理对象接口向它发送进程间通信请求。

4.5.2 具体实现

在 Android 系统中，通常通过 frameworks/base/services/java/com/android/server 目录中的文件实现硬件访问服务。因此把实现了硬件访问服务 FregService 的文件 FregService.java 也保存在这个目录中，其具体内容如下所示。

```
package com.android.server;

import android.content.Context;
import android.os.IFregService;
import android.util.Slog;

public class FregService extends IFregService.Stub {
private static final String TAG = "FregService";

private int mPtr = 0;
```

```

FregService() {
mPtr = init_native();

if(mPtr == 0) {
Slog.e(TAG, "Failed to initialize freg service.");
}
}

public void setVal(int val) {
if(mPtr == 0) {
Slog.e(TAG, "Freg service is not initialized.");
return;
}

setVal_native(mPtr, val);
}

public int getVal() {
if(mPtr == 0) {
Slog.e(TAG, "Freg service is not initialized.");
return 0;
}

return getVal_native(mPtr);
}

private static native int init_native();
private static native void setVal_native(int ptr, int val);
private static native int getVal_native(int ptr);
};
```

在上述代码中，硬件访问服务 FregService 继承了类 IFregService.Stub，并且实现了 IFregService 接口的成员函数 setVal()和 getVal()。其中，成员函数 setVal 通过调用 JNI 方法 setVal_native 来写虚拟硬件设备 freg 的寄存器 val，而成员函数 getVal()调用 JNI 方法 getVal_native()来读虚拟硬件设备 freg 的寄存器 val。在启动硬件访问服务 FregService 时，会通过调用 JNI 函数 init_native()来打开虚拟硬件设备 freg，并且获得它的一个句柄值，保存在成员变量 mPtr 中。如果硬件访问服务 FregService 打开虚拟硬件设备 freg 失败，那么它的成员变量 mPtr 的值就等于 0；否则，就得到一个大于 0 的句柄值。这个句柄值实际上是指向虚拟硬件设备 freg 在硬件抽象层中的一个设备对象，硬件访问服务 FregService 的成员函数 setVal()和 getVal()在访问虚拟硬件设备 freg 的寄存器 val 时，必须要指定这个句柄值，以便硬件访问服务 FregService 的 JNI 实现可以知道它所要访问的是哪一个硬件设备。

4.6 分析 Mokoid 实例

Google 针对 HAL 提供了一个官方实例工程 Mokoid，在此工程中提供了一个 LedTest 演示程序，此程序实例完整地演示了 Android 层次结构和 HAL 架构编程的方法和流程。本节将详细分析 Mokoid 工程的基本源码。

4.6.1 获取实例工程源码

读者可以从网络中获取 LedTest 示例程序的源码，方法是在 Linux 中使用下面的下载命令。

```
#svn checkout http://mokoid.googlecode.com/svn/trunk/mokoid-read-only
```

下载 Mokoid 工程文件后，其目录结构如图 4-6 所示。

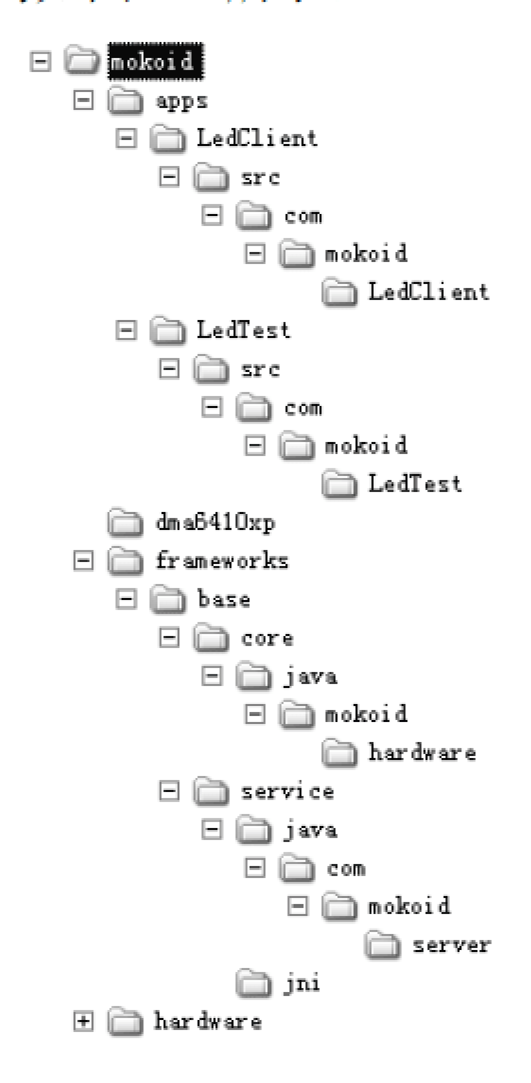


图 4-6 Mokoid 工程的目录结构

Mokoid 工程代码树的具体说明如下。

```
.
|-- apps   -- 测试应用程序
|    |-- LedClient -- 直接调用 service 控制硬件
|    |    |-- AndroidManifest.xml
|    |    `-- src
|    |         `-- com
|    |              `-- mokoid
|    |                   `-- LedClient
|    |                        `-- LedClient.java
|    `-- LedTest -- 通过 manager 来控制硬件
|         |-- AndroidManifest.xml
|         `-- src
|              `-- com
|                   `-- mokoid
|                        `-- LedTest
|                             |-- LedSystemServer.java
|                             `-- LedTest.java
|-- frameworks -- 框架代码
|    `-- base
|         |-- core
|         |    `-- java
|         |         `-- mokoid
|         |              `-- hardware
```

```
|           |                      |-- ILedService.aidl -- Android Interface Definition Language 代码，提供 LedService 的接口
|           |                      `-- LedManager.java -- LedManager 实现代码
|           `-- service
|                |-- com.mokoid.server.xml
|                |-- java
|                |    `-- com
|                |         `-- mokoid
|                |              `-- server
|                |                   `-- LedService.java -- LedService 的 Java 实现代码
|                `-- jni
|                     `-- com_mokoid_server_LedService.cpp -- LedService 的 JNI 实现代码
|-- hardware
    `-- modules
        |-- include
        |    `-- mokoid
        |         `-- led.h
        `-- led
             `-- led.c -- led 实际控制硬件的代码
```

在 Android 系统中需要通过 JNI（Java Native Interface）实现 HAL，因为 JNI 是 Java 程序可以调用 C/C++编写的动态链接库，所以可以使用 C/C++语言编写 HAL，这样做的好处是拥有更高的效率。在 Android 系统中有如下两种访问 HAL 的方式。

（1）Android APP 直接通过 service 调用.so 格式的 JNI，这种方式虽然比较简单高效，但是不正规。

（2）经过 Manager 调用 Service，此方式实现起来比较复杂，但是更符合目前的 Android 框架。在此方法中，在进程 LegManager 和 LedService（Java）中需要通过进程通信的方式实现通信。

在 Mokoid 工程中分别实现了上述两种方法，下面将详细介绍这两种方法的具体实现原理。

4.6.2　直接调用 service 方法的实现代码

（1）HAL 层的实现代码

文件 hardware/modules/led/led.c 的实现代码如下所示。

```
#define LOG_TAG "MokoidLedStub"
#include <hardware/hardware.h>
#include <fcntl.h>
#include <errno.h>
#include <cutils/log.h>
#include <cutils/atomic.h>
#include <mokoid/led.h>
/*************************************************************************************/
int led_device_close(struct hw_device_t* device)
{
	struct led_control_device_t* ctx = (struct led_control_device_t*)device;
	if (ctx) {
		free(ctx);
	}
	return 0;
}
int led_on(struct led_control_device_t *dev, int32_t led)
{
```

```
        LOGI("LED Stub: set %d on.", led);
        return 0;
}
int led_off(struct led_control_device_t *dev, int32_t led)
{
        LOGI("LED Stub: set %d off.", led);
        return 0;
}
static int led_device_open(const struct hw_module_t* module, const char* name,
            struct hw_device_t** device)
{
        struct led_control_device_t *dev;
        dev = (struct led_control_device_t *)malloc(sizeof(*dev));
        memset(dev, 0, sizeof(*dev));
        dev->common.tag =  HARDWARE_DEVICE_TAG;
        dev->common.version = 0;
        dev->common.module = module;
        dev->common.close = led_device_close;
        dev->set_on = led_on;//实例化支持的操作
        dev->set_off = led_off;
        *device = &dev->common;//将实例化的 led_control_device_t 地址返回给 JNI 层
success:
        return 0;
}
static struct hw_module_methods_t led_module_methods = {
       open: led_device_open
};
const struct led_module_t HAL_MODULE_INFO_SYM = {
    //定义此对象相当于向系统注册了一个 ID 为 LED_HARDWARE_MODULE_ID 的 stub
  common: {
            tag: HARDWARE_MODULE_TAG,
            version_major: 1,
            version_minor: 0,
            id: LED_HARDWARE_MODULE_ID,
            name: "Sample LED Stub",
            author: "The Mokoid Open Source Project",
            methods: &led_module_methods,//实现了一个 open 的方法供 JNI 层调用
       }
       /* supporting APIs go here */
};
```

（2）JNI 层的实现代码

文件 frameworks/base/service/jni/com_mokoid_server_LedService.cpp 的实现代码如下所示。

```
struct led_control_device_t *sLedDevice = NULL;
static jboolean mokoid_setOn(JNIEnv* env, jobject thiz, jint led) {
      LOGI("LedService JNI: mokoid_setOn() is invoked.");
      if (sLedDevice == NULL) {
            LOGI("LedService JNI: sLedDevice was not fetched correctly.");
            return -1;
      } else {
            return sLedDevice->set_on(sLedDevice, led);//调用 HAL 层的注册方法
```

```
    }
}
static jboolean mokoid_setOff(JNIEnv* env, jobject thiz, jint led) {
    LOGI("LedService JNI: mokoid_setOff() is invoked.");
    if (sLedDevice == NULL) {
        LOGI("LedService JNI: sLedDevice was not fetched correctly.");
        return -1;
    } else {
        return sLedDevice->set_on(sLedDevice, led); //调用 HAL 层的注册方法
    }
}
/** helper APIs——JNI 通过 LED_HARDSOFTWARE_MODULE_ID 找到对应的 stub*/
static inline int led_control_open(const struct hw_module_t* module,
        struct led_control_device_t** device) {
    return module->methods->open(module,
            LED_HARDWARE_MODULE_ID, (struct hw_device_t**)device);
}
static jboolean
mokoid_init(JNIEnv *env, jclass clazz)
{
    led_module_t* module;
    if (hw_get_module(LED_HARDWARE_MODULE_ID, (const hw_module_t**)&module) == 0) {
        LOGI("LedService JNI: LED Stub found.");
        if (led_control_open(&module->common, &sLedDevice) == 0) {
         LOGI("LedService JNI: Got Stub operations.");
            return 0;
        }
    }
    LOGE("LedService JNI: Get Stub operations failed.");
    return -1;
}
//JNINativeMethod 是 JNI 层的注册方法
static const JNINativeMethod gMethods[] = {
    {"_init",            "()Z",//Framework 层调用_init 时触发
              (void*)mokoid_init},
    { "_set_on",              "(I)Z",
                         (void*)mokoid_setOn },
    { "_set_off",             "(I)Z",
                         (void*)mokoid_setOff },
};
static int registerMethods(JNIEnv* env) {
    static const char* const kClassName =
        "com/mokoid/server/LedService";
    jclass clazz;
    /*  寻找类 class */
    clazz = env->FindClass(kClassName);
    if (clazz == NULL) {
        LOGE("Can't find class %s\n", kClassName);
        return -1;
    }
    /* register all the methods */
```

```
    if (env->RegisterNatives(clazz, gMethods,
            sizeof(gMethods) / sizeof(gMethods[0])) != JNI_OK)
    {
        LOGE("Failed registering methods for %s\n", kClassName);
        return -1;
    }
    return 0;
}
// ---------------------------------------------------------------------------
////Framework 层加载 JNI 库时调用
jint JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = NULL;
    jint result = -1;
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: GetEnv failed\n");
        goto bail;
    }
    assert(env != NULL);
    if (registerMethods(env) != 0) {
        LOGE("ERROR: PlatformLibrary native registration failed\n");
        goto bail;
    }
    /* success -- return valid version number */
    result = JNI_VERSION_1_4;
bail:
    return result;
}
```

（3）Service 的实现代码

这里的 Service 属于 Framework 层，实现文件是 LedService.java，保存在 frameworks/base/service/java/com/mokoid/server 目录中。

LedService.java 的具体实现代码如下所示。

```
package com.mokoid.server;

import android.util.Config;
import android.util.Log;
import android.content.Context;
import android.os.Binder;
import android.os.Bundle;
import android.os.RemoteException;
import android.os.IBinder;
import mokoid.hardware.ILedService;

public final class LedService extends ILedService.Stub {

    static {
        System.load("/system/lib/libmokoid_runtime.so");      //加载 JNI 动态库
    }

    public LedService() {
        Log.i("LedService", "Go to get LED Stub...");
```

```
        _init();
    }

    /*
     * Mokoid LED 本地方法
     */
    public boolean setOn(int led) {
        Log.i("MokoidPlatform", "LED On");
     return _set_on(led);
    }
    public boolean setOff(int led) {
        Log.i("MokoidPlatform", "LED Off");
     return _set_off(led);
    }
    private static native boolean _init();                  //声明 JNI 库可以提供的方法
    private static native boolean _set_on(int led);
    private static native boolean _set_off(int led);
}
```

（4）编写测试应用程序

测试应用程序属于 APP 层，文件 apps/LedClient/src/com/mokoid/LedClient/LedClient.java 的实现代码如下所示。

```
package com.mokoid.LedClient;
import com.mokoid.server.LedService;                     //导入 Framework 层的 LedService

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class LedClient extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Call an API on the library.
     LedService ls = new LedService();                   //实例化 LedService
     ls.setOn(1);                                        //通过 LedService 提供的方法控制底层硬件

        TextView tv = new TextView(this);
        tv.setText("LED 0 is on.");
        setContentView(tv);
    }
}
```

4.6.3　通过 Manager 调用 service 的实现代码

（1）实现 Manager

应用程序通过 Manager 和 Service 来实现通信功能，文件 frameworks/base/core/java/mokoid/hardware/LedManager.java 的实现代码如下所示。

```
package mokoid.hardware;

import android.content.Context;
import android.os.Binder;
import android.os.Bundle;
import android.os.Parcelable;
import android.os.ParcelFileDescriptor;
import android.os.Process;
import android.os.RemoteException;
import android.os.Handler;
import android.os.Message;
import android.os.ServiceManager;
import android.util.Log;
import mokoid.hardware.ILedService;

public class LedManager
{
    private static final String TAG = "LedManager";
    private ILedService mLedService;

    public LedManager() {

        mLedService = ILedService.Stub.asInterface(
                                ServiceManager.getService("led"));

     if (mLedService != null) {
            Log.i(TAG, "The LedManager object is ready.");
     }
      }

      public boolean LedOn(int n) {
        boolean result = false;

        try {
            result = mLedService.setOn(n);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in LedManager.LedOn:", e);
        }
        return result;
    }

    public boolean LedOff(int n) {
        boolean result = false;

        try {
            result = mLedService.setOff(n);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in LedManager.LedOff:", e);
        }
        return result;
    }
}
```

因为 LedService 和 LedManager 分别属于不同的进程，所以在此需要考虑不同进程之间的通信问题。此时在 Manager 中可以增加一个 aidl 文件来描述通信接口，文件 frameworks/base/core/java/mokoid/hardware/ILedService.aidl 的实现代码如下所示。

```
package mokoid.hardware;

interface ILedService
{
    boolean setOn(int led);
    boolean setOff(int led);
}
```

（2）实现 SystemServer

SystemServer 属于 APP 层，文件 apps/LedTest/src/com/mokoid/LedTest/LedSystemServer.java 的主要实现代码如下所示。

```
package com.mokoid.LedTest;

import com.mokoid.server.LedService;

import android.os.IBinder;
import android.os.ServiceManager;
import android.util.Log;
import android.app.Service;
import android.content.Context;
import android.content.Intent;

public class LedSystemServer extends Service {
//代表一个后台进程
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    public void onStart(Intent intent, int startId) {
        Log.i("LedSystemServer", "Start LedService...");
     /* Please also see SystemServer.java for your interests. */
     LedService ls = new LedService();
        try {
            ServiceManager.addService("led", ls);
        } catch (RuntimeException e) {
            Log.e("LedSystemServer", "Start LedService failed.");
        }
    }
}
```

（3）APP 测试程序

此处的测试程序属于 APP 层，文件 mokoid-read-only/apps/LedTest/src/com/mokoid/LedTest/LedTest. java 的实现代码如下所示。

```
package com.mokoid.LedTest;
import mokoid.hardware.LedManager;
import com.mokoid.server.LedService;

import android.app.Activity;
```

```
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;
import android.widget.Button;
import android.content.Intent;
import android.view.View;

public class LedTest extends Activity implements View.OnClickListener {
      private LedManager mLedManager = null;

      @Override
      public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            startService(new Intent("com.mokoid.systemserver"));
            Button btn = new Button(this);
            btn.setText("Click to turn LED 1 On");
       btn.setOnClickListener(this);
            setContentView(btn);
      }
      public void onClick(View v) {
               if (mLedManager == null) {
             Log.i("LedTest", "Creat a new LedManager object.");
             mLedManager = new LedManager();
               }
               if (mLedManager != null) {
             Log.i("LedTest", "Got LedManager object.");
       }
               mLedManager.LedOn(1);
               TextView tv = new TextView(this);
               tv.setText("LED 1 is On.");
               setContentView(tv);
      }
}
```

4.7　HAL 和系统移植

Android 的硬件抽象层和系统移植密切相关，本节将详细讲解移植 Android 5.0 HAL 的基本知识，为读者学习本书后面的知识打下基础。

4.7.1　移植各个 Android 部件的方式

在 Android 系统中，不同子系统的移植方法不同。不同部件的移植方式如下。

☑　显示系统：使用 Framebuffer 标准或其他驱动程序，对应的硬件抽象层是 Gralloc。
☑　用户输入系统：使用 Event 设备的驱动程序，对应的硬件抽象层是 EventHub。
☑　3D 加速系统：使用非标准的驱动程序，对应的硬件抽象层是 OpenGL。
☑　音频系统：使用非标准的驱动程序，对应的是 C++继承的硬件抽象层。

- ☑ 视频输出系统：使用非标准的驱动程序，对应的硬件抽象层是 overlay 模块。
- ☑ 摄像头系统：使用非标准的驱动程序，对应的是 C++继承的硬件抽象层。
- ☑ 多媒体解码系统：使用非标准的驱动程序，对应的硬件抽象层是 Skia 和 OpenMax 插件。
- ☑ 电话系统：使用非标准的驱动程序，对应的硬件抽象层是动态开发插件库。
- ☑ GPS 定位系统：使用非标准的驱动程序，对应的硬件抽象层通常是直接接口。
- ☑ 无线局域网：使用 Wlan 驱动程序，对应的硬件抽象层分别是 Linux 下的 Wpa 和 Android 下的 WiFi。
- ☑ 蓝牙系统：使用 Bluetooth 驱动程序，对应的硬件抽象层分别是 Linux 下的 Bluez 和 Android 下的 Bluedroid。
- ☑ 传感器系统：使用非标准的驱动程序，对应的硬件抽象层是 Sensor 硬件模块。
- ☑ 振动器系统：使用 Sys 文件系统中固定位置的驱动程序，对应的硬件抽象层是 Android 标准的直接接口。
- ☑ 背光和指示灯系统：使用非标准的驱动程序，对应的硬件抽象层是 Light 硬件模块。
- ☑ 警告器系统：使用 Misc 驱动程序，对应的硬件抽象层是 Android 标准的 JNI 层。
- ☑ 电池管理系统：使用 Sys 文件系统中固定位置的驱动程序，对应的硬件抽象层是 Android 标准的直接接口。

4.7.2　设置设备权限

当 Android 系统启动时，在内核引导参数上一般都会设置 init=/init，此时如果内核成功挂载了这个文件系统之后，首先运行的就是这个根目录下的 init 程序。这个 init 程序是 Android 系统运行后的第一个用户空间的程序，它以守护进程的方式运行。

当我们需要增加驱动程序的设备节点时，需要随之更改这些设备节点的属性，这些更改内容被保存在文件 system/core/init/devices.c 中。此文件代码比较冗长，接下来将只对和权限有关的代码进行讲解。

- ☑ 定义 perms_表示设备的类型，具体代码如下所示。

```
struct perms_ {
    char *name;
    mode_t perm;
    unsigned int uid;
    unsigned int gid;
    unsigned short prefix;
};
```

- ☑ 定义数组 devperms 表示系统中的设备，具体代码如下所示。

```
static struct perms_ devperms[] = {
    { "/dev/null",          0666,   AID_ROOT,       AID_ROOT,       0 },
    { "/dev/zero",          0666,   AID_ROOT,       AID_ROOT,       0 },
    { "/dev/full",          0666,   AID_ROOT,       AID_ROOT,       0 },
    { "/dev/ptmx",          0666,   AID_ROOT,       AID_ROOT,       0 },
    { "/dev/tty",           0666,   AID_ROOT,       AID_ROOT,       0 },
    { "/dev/random",        0666,   AID_ROOT,       AID_ROOT,       0 },
    { "/dev/urandom",       0666,   AID_ROOT,       AID_ROOT,       0 },
    { "/dev/ashmem",        0666,   AID_ROOT,       AID_ROOT,       0 },
    { "/dev/binder",        0666,   AID_ROOT,       AID_ROOT,       0 },

        /* logger should be world writable (for logging) but not readable */
    { "/dev/log/",          0662,   AID_ROOT,       AID_LOG,        1 },
```

```
        /* these should not be world writable */
    { "/dev/android_adb",          0660,   AID_ADB,           AID_ADB,           0 },
    { "/dev/android_adb_enable",   0660,   AID_ADB,           AID_ADB,           0 },
    { "/dev/ttyMSM0",              0600,   AID_BLUETOOTH,     AID_BLUETOOTH,     0 },
    { "/dev/ttyHS0",               0600,   AID_BLUETOOTH,     AID_BLUETOOTH,     0 },
    { "/dev/uinput",               0600,   AID_BLUETOOTH,     AID_BLUETOOTH,     0 },
    { "/dev/alarm",                0664,   AID_SYSTEM,        AID_RADIO,         0 },
    { "/dev/tty0",                 0660,   AID_ROOT,          AID_SYSTEM,        0 },
    { "/dev/graphics/",            0660,   AID_ROOT,          AID_GRAPHICS,      1 },
    { "/dev/hw3d",                 0660,   AID_SYSTEM,        AID_GRAPHICS,      0 },
    { "/dev/input/",               0660,   AID_ROOT,          AID_INPUT,         1 },
    { "/dev/eac",                  0660,   AID_ROOT,          AID_AUDIO,         0 },
    { "/dev/cam",                  0660,   AID_ROOT,          AID_CAMERA,        0 },
    { "/dev/pmem",                 0660,   AID_SYSTEM,        AID_GRAPHICS,      0 },
    { "/dev/pmem_gpu",             0660,   AID_SYSTEM,        AID_GRAPHICS,      1 },
    { "/dev/pmem_adsp",            0660,   AID_SYSTEM,        AID_AUDIO,         1 },
    { "/dev/pmem_camera",          0660,   AID_SYSTEM,        AID_CAMERA,        1 },
    { "/dev/oncrpc/",              0660,   AID_ROOT,          AID_SYSTEM,        1 },
    { "/dev/adsp/",                0660,   AID_SYSTEM,        AID_AUDIO,         1 },
    { "/dev/mt9t013",              0660,   AID_SYSTEM,        AID_SYSTEM,        0 },
    { "/dev/akm8976_daemon",       0640,   AID_COMPASS,       AID_SYSTEM,        0 },
    { "/dev/akm8976_aot",          0640,   AID_COMPASS,       AID_SYSTEM,        0 },
    { "/dev/akm8976_pffd",         0640,   AID_COMPASS,       AID_SYSTEM,        0 },
    { "/dev/msm_pcm_out",          0660,   AID_SYSTEM,        AID_AUDIO,         1 },
    { "/dev/msm_pcm_in",           0660,   AID_SYSTEM,        AID_AUDIO,         1 },
    { "/dev/msm_pcm_ctl",          0660,   AID_SYSTEM,        AID_AUDIO,         1 },
    { "/dev/msm_snd",              0660,   AID_SYSTEM,        AID_AUDIO,         1 },
    { "/dev/msm_mp3",              0660,   AID_SYSTEM,        AID_AUDIO,         1 },
    { "/dev/msm_audpre",           0660,   AID_SYSTEM,        AID_AUDIO,         0 },
    { "/dev/htc-acoustic",         0660,   AID_SYSTEM,        AID_AUDIO,         0 },
    { "/dev/smd0",                 0640,   AID_RADIO,         AID_RADIO,         0 },
    { "/dev/qmi",                  0640,   AID_RADIO,         AID_RADIO,         0 },
    { "/dev/qmi0",                 0640,   AID_RADIO,         AID_RADIO,         0 },
    { "/dev/qmi1",                 0640,   AID_RADIO,         AID_RADIO,         0 },
    { "/dev/qmi2",                 0640,   AID_RADIO,         AID_RADIO,         0 },
    { NULL, 0, 0, 0, 0 },
};
```

在上述数组中分别设置了设备的权限、所属用户和所属组，各个权限值的含义和 Linux 中的完全一致。3 个数组分别表示所属用户、所属组和其他人的权限，其中 4 表示可读，2 表示可写，1 表示可执行。例如数组内的首行代码如下所示。

```
{ "/dev/null", 0666, AID_ROOT, AID_ROOT, 0 },
```

/dev/null 是一个标准的设备，其权限是 0666，表示任何用户可以对其进行读写操作。如果需要增加一个新的设备节点文件，需要在数组 devperms 中新增加一行内容。

☑ 两个函数

在文件中有两个比较重要的函数：handle_device_event()和 make_device()，具体代码如下所示。

```
static void handle_device_event(struct uevent *uevent)
{
    ...
    /* are we block or char? where should we live? */
```

```
    if(!strncmp(uevent->path, "/block", 6)) {
        block = 1;
        base = "/dev/block/";//根据 uevent 路径改变该节点路径
        mkdir(base, 0755);
    } else {
        block = 0;
            /* this should probably be configurable somehow */
        if(!strncmp(uevent->path, "/class/graphics/", 16)) {
            base = "/dev/graphics/";//根据 uevent 路径改变该  uevent 需要创建节点的路径
            mkdir(base, 0755);
        } else if (!strncmp(uevent->path, "/class/oncrpc/", 14)) {
            base = "/dev/oncrpc/";
            mkdir(base, 0755);
        } else if (!strncmp(uevent->path, "/class/adsp/", 12)) {
            base = "/dev/adsp/";
            mkdir(base, 0755);
        } else if(!strncmp(uevent->path, "/class/input/", 13)) {
            base = "/dev/input/";//根据 uevent 路径改变该  uevent 需要创建节点的路径
            mkdir(base, 0755);
        } else if(!strncmp(uevent->path, "/class/sensors/", 15)) {
            base = "/dev/sensors/";
            mkdir(base, 0755);
        } else if(!strncmp(uevent->path, "/class/mtd/", 11)) {
            base = "/dev/mtd/";//根据  uevent 路径改变该 uevent 需要创建节点的路径
            mkdir(base, 0755);
        } else if(!strncmp(uevent->path, "/class/misc/", 12) &&
                    !strncmp(name, "log_", 4)) {
            base = "/dev/log/";//根据  uevent 路径改变该 uevent 需要创建节点的路径
            mkdir(base, 0755);
            name += 4;
        } else if(!strncmp(uevent->path, "/class/sound/", 13)) {
            base = "/dev/snd/";
            mkdir(base, 0755);
        } else
            base = "/dev/";
    }

    snprintf(devpath, sizeof(devpath), "%s%s", base, name);

    if(!strcmp(uevent->action, "add")) {
        make_device(devpath, block, uevent->major, uevent->minor);//创建节点文件 devpath
        return;
    }
    ...
}
static void make_device(const char *path, int block, int major, int minor)
{
    unsigned uid;
    unsigned gid;
    mode_t mode;
    dev_t dev;
```

```
    if(major > 255 || minor > 255)
        return;
    mode = get_device_perm(path, &uid, &gid) | (block ? S_IFBLK : S_IFCHR);//获取将要创建的节点是否需要重设它的 mode 数值
    dev = (major << 8) | minor;
    mknod(path, mode, dev);
    chown(path, uid, gid);
}
```

函数 get_device_perm()的功能是验证路径 path 是否和 devperms[]数组中的 inode 路径相同，如果相同则返回 devperms[]数组中指定的 uid、gid 和 mode 数值。这样 make_device 会向/dev 创建 inode 节点，并同时改变该 inode 的 uid 和 gid.[luther.gliethtt]。

☑ 和用户名相关的名称

在文件 system/core/include/private/android_filesystem_config.h 中定义了和用户名相关的名称，其中 android_id_info 表示用户名 ID 的属性。android_id_info 的定义代码如下所示。

```
struct android_id_info {
    const char *name;
    unsigned aid;
};
```

各个用户名 ID 被定义在数组 android_ids[]中，此数组表示了一个映射关系，能够将字符串和整数值对应起来。android_ids[]的定义代码如下所示。

```
static struct android_id_info android_ids[] = {
    { "root",           AID_ROOT, },
    { "system",         AID_SYSTEM, },
    { "radio",          AID_RADIO, },
    { "bluetooth",      AID_BLUETOOTH, },
    { "graphics",       AID_GRAPHICS, },
    { "input",          AID_INPUT, },
    { "audio",          AID_AUDIO, },
    { "camera",         AID_CAMERA, },
    { "log",            AID_LOG, },
    { "compass",        AID_COMPASS, },
    { "mount",          AID_MOUNT, },
    { "wifi",           AID_WIFI, },
    { "dhcp",           AID_DHCP, },
    { "adb",            AID_ADB, },
    { "install",        AID_INSTALL, },
    { "media",          AID_MEDIA, },
    { "shell",          AID_SHELL, },
    { "cache",          AID_CACHE, },
    { "diag",           AID_DIAG, },
    { "net_bt_admin",   AID_NET_BT_ADMIN, },
    { "net_bt",         AID_NET_BT, },
    { "inet",           AID_INET, },
    { "net_raw",        AID_NET_RAW, },
    { "misc",           AID_MISC, },
    { "nobody",         AID_NOBODY, },
};
```

4.7.3　init.rc 初始化

文件 system/core/rootdir/init.rc 可以实现一些简单的初始化操作，Android 5.0 中的启动脚本文件是 init.rc。init.rc 脚本被直接安装到目标系统的根目录下，并被 init 可执行的程序解析。

在 Android 5.0 中，init.rc 是在 init 启动后被执行的启动脚本，主要包含如下内容。

☑　Commands：命令。
☑　Actions：动作。
☑　Triggers：触发条件。
☑　Services：服务。
☑　Options：选项。
☑　Properties：属性。

4.7.4　文件系统的属性

在文件 system/core/include/private/android_filesystem_config.h 中定义了各个文件的属性，其中 fs_path_config 表示文件系统路径的属性，具体定义代码如下所示。

```
struct fs_path_config {
    unsigned mode;              //模式
    unsigned uid;               //用户 ID
    unsigned gid;               //组 ID
    const char *prefix;         //目录前缀
};
```

在数组 android_dirs[]中定义了子目录的属性，定义代码如下所示。

```
static struct fs_path_config android_dirs[] = {
    { 00770, AID_SYSTEM, AID_CACHE,   "cache" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/app" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/app-private" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/dalvik-cache" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/data" },
    { 00771, AID_SHELL,  AID_SHELL,  "data/local/tmp" },
    { 00771, AID_SHELL,  AID_SHELL,  "data/local" },
    { 01771, AID_SYSTEM, AID_MISC,   "data/misc" },
    { 00770, AID_DHCP,   AID_DHCP,   "data/misc/dhcp" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data" },
    { 00750, AID_ROOT,   AID_SHELL,  "sbin" },
    { 00755, AID_ROOT,   AID_SHELL,  "system/bin" },
    { 00755, AID_ROOT,   AID_SHELL,  "system/xbin" },
    { 00777, AID_ROOT,   AID_ROOT,   "system/etc/ppp" }, /* REMOVE */
    { 00777, AID_ROOT,   AID_ROOT,   "sdcard" },
    { 00755, AID_ROOT,   AID_ROOT,   0 },
};
```

然后在数组 android_files[]中定义了默认文件的属性，定义代码如下所示。

```
static struct fs_path_config android_files[] = {
    { 00555, AID_ROOT,      AID_ROOT,      "system/etc/ppp/ip-up" },
    { 00555, AID_ROOT,      AID_ROOT,      "system/etc/ppp/ip-down" },
    { 00440, AID_ROOT,      AID_SHELL,     "system/etc/init.goldfish.rc" },
```

```
    { 00550, AID_ROOT,       AID_SHELL,     "system/etc/init.goldfish.sh" },
    { 00440, AID_ROOT,       AID_SHELL,     "system/etc/init.trout.rc" },
    { 00550, AID_ROOT,       AID_SHELL,     "system/etc/init.ril" },
    { 00550, AID_ROOT,       AID_SHELL,     "system/etc/init.testmenu" },
    { 00550, AID_ROOT,       AID_SHELL,     "system/etc/init.gprs-pppd" },
    { 00550, AID_DHCP,       AID_SHELL,     "system/etc/dhcpcd/dhcpcd-run-hooks" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/dbus.conf" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/bluez/hcid.conf" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/bluez/input.conf" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/bluez/audio.conf" },
    { 00440, AID_RADIO,      AID_AUDIO,     "/system/etc/AudioPara4.csv" },
    { 00644, AID_SYSTEM,     AID_SYSTEM,    "data/app/*" },
    { 00644, AID_SYSTEM,     AID_SYSTEM,    "data/app-private/*" },
    { 00644, AID_APP,        AID_APP,       "data/data/*" },
        /* the following two files are INTENTIONALLY set-gid and not set-uid.
         * Do not change. */
    { 02755, AID_ROOT,       AID_NET_RAW,   "system/bin/ping" },
    { 02755, AID_ROOT,       AID_INET,      "system/bin/netcfg" },
     /* the following four files are INTENTIONALLY set-uid, but they
      * are NOT included on user builds. */
    { 06755, AID_ROOT,       AID_ROOT,      "system/xbin/su" },
    { 06755, AID_ROOT,       AID_ROOT,      "system/xbin/librank" },
    { 06755, AID_ROOT,       AID_ROOT,      "system/xbin/procrank" },
    { 06755, AID_ROOT,       AID_ROOT,      "system/xbin/procmem" },
    { 00755, AID_ROOT,       AID_SHELL,     "system/bin/*" },
    { 00755, AID_ROOT,       AID_SHELL,     "system/xbin/*" },
    { 00750, AID_ROOT,       AID_SHELL,     "sbin/*" },
    { 00755, AID_ROOT,       AID_ROOT,      "bin/*" },
    { 00750, AID_ROOT,       AID_SHELL,     "init*" },
    { 00644, AID_ROOT,       AID_ROOT,       0 },
};
```

4.8 开发自己的 HAL 驱动程序

经过本章前面内容的学习，读者已经了解了 HAL 的具体架构和运行原理，本节将简单演示开发自己的 HAL 的具体过程，并讲解编译调用的方法。

4.8.1 封装 HAL 接口

可以将 Android 的硬件驱动程序看作在内核层，HAL 负责封装硬件驱动，然后再经过 JNI 接口的封装才能给 Java 应用程序调用。封装 HAL 层接口的具体流程如下所示。

（1）在 hardware/libhardware/include/hardware 目录下添加头文件 hello.h，具体方法可以参考当前目录下的头文件 overlay.h。头文件 hello.h 的具体代码如下所示。

```
/*****************************************************
*android_hal_hello_demo
*hello.h
*****************************************************/
```

```
#ifndef ANDROID_HELLO_INTERFACE_H
#define ANDROID_HELLO_INTERFACE_H
#include <hardware/hardware.h>
__BEGIN_DECLS
#define HELLO_HARDWARE_MODULE_ID "hello"
struct hello_module_t {
struct hw_module_t common;
};
struct hello_device_t {
struct hw_device_t common;
int fd;
int (*get_val)(struct hello_device_t *dev,int val);
int (*set_val)(struct hello_device_t *dev,int val);
};
__END_DECLS;
#endif
```

（2）在 hardware/libhardware/modules 目录下新建一个名为 hello 的文件夹，并在此文件夹中新建 hello.c 和 Android.mk 文件，具体方法读者可以参考 modules/overlay 目录下的内容。文件 hello.c 的具体代码如下所示。

```
/********************************************************************************
*android_hal_hello_demo
*hello.c
********************************************************************************/
#include <hardware/hardware.h>
#include <hardware/hello.h>
#include <fcntl.h>
#include <errno.h>
#include <cutils/log.h>
#include <cutils/atomic.h>
#define LOG_TAG "hello_stub"
#define DEVICE_NAME "/dev/hello"
#define MODULE_NAME "Hello"
#define MODULE_AUTHOR "729017304@qq.com"
static int hello_device_open(const struct hw_module_t *module,const char *name,struct hw_device_t** device);
static int hello_device_close(struct hw_device_t* device);
static int hello_set_val(struct hello_device_t*dev,int val);
static int hello_get_val(struct hello_device_t *dev,int *val);
static struct hw_module_methods_t hello_module_methods   = {
open : hello_device_open
};

struct hello_module_t HAL_MODULE_INFO_SYM = {
common: {
tag: HARDWARE_MODULE_TAG,
version_major: 1,
version_minor: 0,
id: HELLO_HARDWARE_MODULE_ID,
name: MODULE_NAME,
author: MODULE_AUTHOR,
methods: &hello_module_methods,
}
```

```
};

static int hello_device_open(const struct hw_module_t* module, const char* name, struct hw_device_t** device)
{
struct hello_device_t* dev;
dev = (struct hello_device_t*)malloc(sizeof(struct hello_device_t));
if(!dev) {
LOGE("Hello Stub: failed to alloc space");
return -EFAULT;
}
memset(dev, 0, sizeof(struct hello_device_t));
dev->common.tag = HARDWARE_DEVICE_TAG;
dev->common.version = 0;
dev->common.module = (hw_module_t*)module;
dev->common.close = hello_device_close;
dev->set_val = hello_set_val;
dev->get_val = hello_get_val;
if((dev->fd = open(DEVICE_NAME, O_RDWR)) == -1) {
LOGE("Hello Stub: failed to open /dev/hello -- %s.", strerror(errno));free(dev);
   return -EFAULT;
}

  *device = &(dev->common);
LOGI("Hello Stub: open /dev/hello successfully.");
return 0;
}
static int hello_device_close(struct hw_device_t* device) {
struct hello_device_t* hello_device = (struct hello_device_t*)device;
if(hello_device) {
close(hello_device->fd);
free(hello_device);
}
return 0;
}
static int hello_set_val(struct hello_device_t* dev, int val) {
LOGI("Hello Stub: set value %d to device.", val);
write(dev->fd, &val, sizeof(val));
return 0;
}
static int hello_get_val(struct hello_device_t* dev, int* val) {
if(!val) {
LOGE("Hello Stub: error val pointer");
return -EFAULT;
}
read(dev->fd, val, sizeof(*val));
LOGI("Hello Stub: get value %d from device", *val);
return 0;
}
```

Android.mk 文件的具体代码如下所示。

```
/*********************************************************************************
*android_hal_hello_demo
```

```
*Android.mk
*****************************************************************************/
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := hello.c
LOCAL_MODULE := hello.default
include $(BUILD_SHARED_LIBRARY)
```

4.8.2　开始编译

因为在获取源码后已经 make 编译过一次 Android 源码了，所以此时就不需要重新 make clean 并编译了，只需将模块编译好，然后再执行 make snod 命令即可将新的模块编译到镜像中。在 Android 源码的 build 目录下有一个配置环境的脚本文件 envsetup.sh，在此文件中包含了编译工具 m、mm 和 mmm。在此使用工具 mmm 进行编译。

（1）在 Android 源码包中执行如下所示的命令。

```
[root@localhost Android-5.0]# sh build/envsetup.sh
[root@localhost Android-5.0]#croot
```

（2）然后使用 mmm 工具来编译模块，具体命令如下所示。

```
[root@localhost Android-5.0]#mmm hardware/libhardware/modules/hello
```

如果出现找不到 liblog.so 库文件的错误，则需要编译生成 liblog.so 这个库文件来解决。

（3）编译生成 liblog.so，具体命令如下所示。

```
[root@localhost Android-5.0]#make liblog
```

（4）接下来开始重新编译，具体命令如下所示。

```
[root@localhost Android-5.0]#mmm hardware/libhardware/modules/hello
```

最终会成功看到生成库文件 hello.default.so，接下来需要重新打包镜像，具体命令如下所示。

```
[root@localhost Android-5.0]make snod
```

（5）最后的工作是重新封装 JNI，读者可以参阅其他相关的资料。

第 5 章　Binder 通信驱动详解

在 Android 系统中，应用程序都是由 Activity 和 Service 组成的。Service 通常运行在独立的进程中，而 Activity 既可运行在同一个进程中，也可运行在不同的进程中。不在同一个进程中的 Activity 或 Service 是如何通过 Binder 机制实现进程间的通信功能的呢？Binder 是 Android 系统提供的一种 IPC（进程间通信）机制。由于 Android 是基于 Linux 内核的，因此，除了 Binder 以外，还存在其他的 IPC 机制，如管道和 socket 等。Binder 相对于其他 IPC 机制来说，就更加灵活和方便了。Binder 驱动程序的代码在 kernel/drivers/staing/android/binder.c 目录中保存，另外该目录下还有一个 binder.h 头文件。Binder 是一个虚拟设备，所以它的代码相比而言还算简单，读者只要有基本的 Linux 驱动开发方面的知识就能读懂它。/proc/binder 目录下的内容可用来查看 Binder 设备的运行状况。本章将详细分析 Android 的进程通信机制 Binder 驱动程序的实现源码，为读者学习本书后面的知识打下基础。

5.1　分析 Binder 驱动程序

可以基本上将 Android 系统看作是一个基于 Binder 通信的 C/S 架构。Binder 就像网络一样，把系统的各个部分连接在了一起。在基于 Binder 通信的 C/S 架构体系中，除了 C/S 架构所包括的 Client 端和 Server 端外，Android 系统还有一个全局的 ServiceManager 端，其作用是管理系统中的各种服务（Service）。

Binder 采用 AIDL（Android Interface Description Language）来描述进程间通信的接口。Binder 作为一个特殊的字符设备，其设备节点是/dev/binder，主要驱动程序代码在 kernel/drivers/staging/binder.h 和 kernel/drivers/staging/binder.c 文件中实现。

本节将详细分析上述驱动文件的实现源码。

5.1.1　数据结构 binder_work

数据结构 binder_work 表示在 binder 驱动中进程所要处理的工作项，定义代码如下所示。

```
struct binder_work {
	struct list_head entry;
	enum {
		BINDER_WORK_TRANSACTION = 1,
		BINDER_WORK_TRANSACTION_COMPLETE,
		BINDER_WORK_NODE,
		BINDER_WORK_DEAD_BINDER,
		BINDER_WORK_DEAD_BINDER_AND_CLEAR,
		BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
	} type;
};
```

在上述结构体定义中，entry 被定义为 list_head 类型，用于实现一个双向链表，能够存储所有 binder_ work 的队列；并且还包含了一个 enum 类型的 type：binder_work。

5.1.2　结构体 binder_node

结构体 binder_node 用来定义 Binder 实体对象。在 Android 系统中，每一个 Service 组件在 Binder 驱动程序中都有一个 Binder 实体对象。定义 binder_node 的代码如下所示。

```
struct binder_node {
	int debug_id;
	struct binder_work work;
	union {
		struct rb_node rb_node;
		struct hlist_node dead_node;
	};
	struct binder_proc *proc;
	struct hlist_head refs;
	int internal_strong_refs;
	int local_weak_refs;
	int local_strong_refs;
	void __user *ptr;
	void __user *cookie;
	unsigned has_strong_ref:1;
	unsigned pending_strong_ref:1;
	unsigned has_weak_ref:1;
	unsigned pending_weak_ref:1;
	unsigned has_async_transaction:1;
	unsigned accept_fds:1;
	unsigned min_priority:8;
	struct list_head async_todo;
};
```

驱动中的 Binder 实体也叫做“节点”，隶属于提供实体的进程。结构体 binder_node 中各个成员的具体说明如表 5-1 所示。

表 5-1　结构体 binder_node 中的成员说明信息

成　　员	含　　义
int debug_id;	用于调试
struct binder_work work;	当本节点引用计数发生改变，需要通知所属进程时，通过该成员挂入所属进程的 to-do 队列中，唤醒所属进程执行 Binder 实体引用计数的修改

union {
struct rb_node rb_node;
struct hlist_node dead_node;
};
	每个进程都维护一棵红黑树，以 Binder 实体在用户空间的指针，即本结构的 ptr 成员为索引存放该进程所有的 Binder 实体。这样驱动可以根据 Binder 实体在用户空间的指针很快找到其位于内核的节点。rb_node 用于将本节点链入该红黑树中
	销毁节点时须将 rb_node 从红黑树中摘除，但如果本节点还有引用没有切断，就用 dead_node 将节点隔离到另一个链表中，直到通知所有进程切断与该节点的引用后，该节点才可能被销毁

struct binder_proc *proc;	本成员指向节点所属的进程，即提供该节点的进程
struct hlist_head refs;	本成员是队列头，所有指向本节点的引用都链接在该队列中。这些引用可能隶属于不同的进程。通过该队列可以遍历指向该节点的所有引用
int internal_strong_refs;	用以实现强指针的计数器：产生一个指向本节点的强引用，该计数就会加 1

续表

成　　员	含　　义
int local_weak_refs;	驱动为传输中的 Binder 设置的弱引用计数。如果一个 Binder 打包在数据包中从一个进程发送到另一个进程，驱动会为该 Binder 增加引用计数，直到接收进程通过 BC_FREE_BUFFER 通知驱动释放该数据包的数据区为止
int local_strong_refs;	驱动为传输中的 Binder 设置的强引用计数
void __user *ptr;	指向用户空间 Binder 实体的指针，来自于 flat_binder_object 的 binder 成员
void __user *cookie;	指向用户空间的附加指针，来自于 flat_binder_object 的 cookie 成员
unsigned has_strong_ref; unsigned pending_strong_ref; unsigned has_weak_ref; unsigned pending_weak_ref	这一组标志用于控制驱动与 Binder 实体所在进程交互式修改引用计数
unsigned has_async_transaction；	该成员表明该节点在 to-do 队列中有异步交互尚未完成。驱动将所有发送往接收端的数据包暂存在接收进程或线程开辟的 to-do 队列中。对于异步交互，驱动做了适当流控：如果 to-do 队列中有异步交互尚待处理则该成员置 1，这将导致新到的异步交互存放在本结构成员– asynch_todo 队列中，而不直接送到 to-do 队列中。目的是为同步交互让路，避免长时间阻塞发送端
unsigned accept_fds	表明节点是否同意接受文件方式的 Binder，来自 flat_binder_object 中 flags 成员的 FLAT_BINDER_FLAG_ACCEPTS_FDS 位。由于接收文件 Binder 会为进程自动打开一个文件，占用有限的文件描述符，节点可以设置该位拒绝这种行为
int min_priority	设置处理 Binder 请求的线程的最低优先级。发送线程将数据提交给接收线程处理时，驱动会将发送线程的优先级也赋予接收线程，使得数据即使跨进程也能以同样优先级得到处理。不过如果发送线程优先级过低，接收线程将以预设的最小值运行 该域的值来自于 flat_binder_object 中的 flags 成员
struct list_head async_todo	异步交互等待队列；用于分流发往本节点的异步交互包

5.1.3　结构体 binder_ref

结构体 binder_ref 用来描述一个 Binder 引用对象，在 Android 系统中，每一个 Client 组件在 Binder 驱动程序中都有一个 Binder 引用对象。定义 binder_ref 的代码如下所示。

```
struct binder_ref {
        int debug_id;
        struct rb_node rb_node_desc;
        struct rb_node rb_node_node;
        struct hlist_node node_entry;
        struct binder_proc *proc;
        struct binder_node *node;
        uint32_t desc;
        int strong;
        int weak;
        struct binder_ref_death *death;
};
```

结构体 binder_ref 中各个成员的具体说明如表 5-2 所示。

表 5-2　结构体 binder_ref 中的成员说明信息

成　员	含　义
int debug_id;	调试用
struct rb_node rb_node_desc;	每个进程有一棵红黑树，进程所有引用以引用号（即本结构的 desc 域）为索引添入该树中。本成员用作链接到该树的一个节点
struct rb_node rb_node_node;	每个进程又有一棵红黑树，进程所有引用以节点实体在驱动中的内存地址（即本结构的 node 域）为索引添入该树中。本成员用作链接到该树的一个节点
struct hlist_node node_entry;	该域将本引用作为节点链入所指向的 Binder 实体结构 binder_node 中的 refs 队列
struct binder_proc *proc;	本引用所属的进程
struct binder_node *node;	本引用所指向的节点（Binder 实体）
uint32_t desc;	本结构的引用号
int strong;	强引用计数
int weak;	弱引用计数
struct binder_ref_death *death;	应用程序向驱动发送 BC_REQUEST_DEATH_NOTIFICATION 或 BC_CLEAR_DEATH_NOTIFICATION 命令，从而当 Binder 实体销毁时能够收到来自驱动的提醒。该域不为空，表明用户订阅了对应实体销毁的“噩耗”

5.1.4　通知结构体 binder_ref_death

binder_ref_death 是一个通知结构体，只要某进程对某 Binder 引用订阅了其实体的死亡通知，那么 binder 驱动将会为该 binder 引用建立一个 binder_ref_death 通知结构体，将其保存在当前进程的对应 Binder 引用结构体的 death 域中。定义 binder_ref_death 的代码如下所示。

```
struct binder_ref_death {
		struct binder_work work;
		void __user *cookie;
};
```

5.1.5　结构体 binder_buffer

结构体 binder_buffer 用来描述一个内核缓冲区，能够在进程之间传输数据。定义 binder_buffer 的代码如下所示。

```
struct binder_buffer {
		struct list_head entry; /* free and allocated entries by address */
		struct rb_node rb_node; /* free entry by size or allocated entry */
						/* by address */
		unsigned free:1;
		unsigned allow_user_free:1;
		unsigned async_transaction:1;
		unsigned debug_id:29;
		struct binder_transaction *transaction;
		struct binder_node *target_node;
		size_t data_size;
		size_t offsets_size;
		uint8_t data[0];
};
```

结构体 binder_buffer 能够存储 Binder 的相关信息，成员的具体说明如下。

☑ entry：构建一个双向链表。
☑ rb_node：表示一个红黑树节点。
☑ transaction：用于中转请求和返回结果。
☑ target_node：是一个目标节点。
☑ data_size：表示数据的大小。
☑ offsets_size：是一个偏移量。
☑ data[0]：用于存储实际数据。

5.1.6 结构体 binder_proc

结构体 binder_proc 表示正在使用 Binder 进程通信机制的进程，能够保存调用 Binder 的各个进程或线程的信息，例如线程 ID、进程 ID、Binder 状态信息等。定义 binder_proc 的具体实现代码如下所示。

```
struct binder_proc {
	//实现双向链表
	struct hlist_node proc_node;
	//线程队列、双向链表、所有的线程信息
	struct rb_root threads;
	struct rb_root nodes;
	struct rb_root refs_by_desc;
	struct rb_root refs_by_node;
	//进程 ID
	int pid;
	struct vm_area_struct *vma;
	struct task_struct *tsk;
	struct files_struct *files;
	struct hlist_node deferred_work_node;
	int deferred_work;
	void *buffer;
	ptrdiff_t user_buffer_offset;

	struct list_head buffers;
	struct rb_root free_buffers;
	struct rb_root allocated_buffers;
	size_t free_async_space;

	struct page **pages;
	size_t buffer_size;
	uint32_t buffer_free;
	struct list_head todo;
	//等待队列
	wait_queue_head_t wait;
	//Binder 状态
	struct binder_stats stats;
	struct list_head delivered_death;
	//最大线程
	int max_threads;
	int requested_threads;
```

```
    int requested_threads_started;
    int ready_threads;
    //默认优先级
    long default_priority;
};
```

在上述代码中，成员 proc_node 用于实现双向链表，成员 threads 用于存储所有的线程信息。

5.1.7　结构体 binder_thread

结构体 binder_thread 用于存储每一个单独的线程的信息，表示 Binder 线程池中的一个线程。定义 binder_thread 的具体实现代码如下所示。

```
struct binder_thread {
        struct binder_proc *proc;
        struct rb_node rb_node;
        int pid;
        int looper;
        struct binder_transaction *transaction_stack;
        struct list_head todo;
        uint32_t return_error;
        uint32_t return_error2;
        wait_queue_head_t wait;
        struct binder_stats stats;
};
```

各个成员的具体说明如下。

- ☑ proc：表示当前线程属于哪一个 Binder 进程（binder_proc 指针）。
- ☑ rb_node：是一个红黑树节点。
- ☑ pid：表示线程的 pid。
- ☑ looper：表示线程的状态信息。
- ☑ transaction_stack：定义了要接收和发送的进程和线程信息，其结构体为 binder_transaction。
- ☑ todo：用于创建一个双向链表。
- ☑ return_error 和 return_error2：表示返回的错误信息代码。
- ☑ wait：是一个等待队列头结构，具体的定义代码如下所示。

```
struct binder_stats {
        int br[_IOC_NR(BR_FAILED_REPLY) + 1];
        int bc[_IOC_NR(BC_DEAD_BINDER_DONE) + 1];
        int obj_created[BINDER_STAT_COUNT];
        int obj_deleted[BINDER_STAT_COUNT];
};
```

各个成员的具体说明如下。

- ☑ br：用来存储 BINDER_WRITE_READ 的写操作命令协议（Binder Driver Return Protocol）。
- ☑ bc：存储着 BINDER_WRITE_READ 的写操作命令协议（Binder Driver Command Protocol）。
- ☑ obj_created：保存 BINDER_STAT_COUNT 的对象计数，当创建一个对象时需要同时调用该成员来增加相应的对象计数，而 obj_deleted 则正好与之相反。

looper 表示的线程状态信息在如下枚举中定义。

```
enum {
        BINDER_LOOPER_STATE_REGISTERED   = 0x01,
```

```
        BINDER_LOOPER_STATE_ENTERED      = 0x02,
        BINDER_LOOPER_STATE_EXITED       = 0x04,
        BINDER_LOOPER_STATE_INVALID      = 0x08,
        BINDER_LOOPER_STATE_WAITING      = 0x10,
        BINDER_LOOPER_STATE_NEED_RETURN = 0x20
};
```

上述枚举主要包括的状态信息有：注册、进入、退出、销毁、等待、需要返回。

5.1.8 结构体 binder_transaction

结构体 binder_transaction 的功能是中转请求和返回结果，并保存接收和要发送的进程信息。定义结构体 binder_transaction 的具体实现代码如下所示。

```
struct binder_transaction {
        int debug_id;//调试相关
        struct binder_work work;
        struct binder_thread *from;
        struct binder_transaction *from_parent;
        struct binder_proc *to_proc;
        struct binder_thread *to_thread;
        struct binder_transaction *to_parent;
        unsigned need_reply : 1;
        struct binder_buffer *buffer;
        unsigned int    code;
        unsigned int    flags;
        long priority;
        long saved_priority;
        uid_t    sender_euid;
};
```

上述成员的具体说明如下。

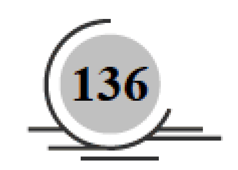
☑ work：是一个 binder_work。
☑ from 和 to_thread：都是一个 binder_thread 对象，用于表示接收和要发送的进程信息。
☑ from_parent 和 to_thread：接收和发送进程信息的父节点。
☑ to_proc：是一个 binder_proc 类型的结构体，还包括 flags、need_reply、优先级（priority）等数据。
☑ sender_euid：Linux 系统中的每个进程都有两个 ID，即用户 ID 和有效用户 ID，UID 一般表示进程的创建者（属于哪个用户创建），EUID 表示进程对于文件和资源的访问权限。sender_euid 表示要发送进程对文件和资源的操作权限。

另外在结构体 binder_transaction 中，还包含类型类 inder_buffer 的一个 buffer，用来表示 binder 的缓冲区信息，inder_buffer 在前面已经进行了讲解。

5.1.9 结构体 binder_write_read

结构体 binder_write_read 的功能是表示在进程之间的通信过程中传输的数据，数据包中有一个 cmd 域用于区分不同的请求。定义结构体 binder_write_read 的实现代码如下所示。

```
struct binder_write_read {
        signed long      write_size;     /* bytes to write */
        signed long      write_consumed; /* bytes consumed by driver */
```



```
        unsigned long    write_buffer;
        signed long      read_size;        /* bytes to read */
        signed long      read_consumed;   /* bytes consumed by driver */
        unsigned long    read_buffer;
};
```

各个成员的具体说明如下。

☑　write_size 和 read_size：分别表示写入和读取的数据的大小。

☑　write_consumed 和 read_consumed：分别表示被消耗的写数据和读数据的大小。

当 Binder 驱动找到处理此事件的进程之后，Binder 驱动就会把需要处理的事件的任务放在读缓冲（binder_write_read）中，返回给这个服务线程，该服务线程则会执行指定命令的操作；处理请求的线程把数据交给合适的对象来执行预定操作，然后把返回结果同样用结构 binder_transaction_data 进行封装，以写命令的方式传回给 Binder 驱动，并将此数据放在一个读缓冲（binder_write_read）中，返回给正在等待结果的原进程（线程），这样就完成了一次通信。

5.1.10　Binder 驱动协议

在 BinderDriverCommandProtocol 中定义了结构体 binder_write_read 包含的命令，具体代码如下所示。

```
enum BinderDriverCommandProtocol {
        BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
        BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
        BC_ACQUIRE_RESULT = _IOW('c', 2, int),
        BC_FREE_BUFFER = _IOW('c', 3, int),
        BC_INCREFS = _IOW('c', 4, int),
        BC_ACQUIRE = _IOW('c', 5, int),
        BC_RELEASE = _IOW('c', 6, int),
        BC_DECREFS = _IOW('c', 7, int),
        BC_INCREFS_DONE = _IOW('c', 8, struct binder_ptr_cookie),
        BC_ACQUIRE_DONE = _IOW('c', 9, struct binder_ptr_cookie),
        BC_ATTEMPT_ACQUIRE = _IOW('c', 10, struct binder_pri_desc),
        BC_REGISTER_LOOPER = _IO('c', 11),
        BC_ENTER_LOOPER = _IO('c', 12),
        BC_EXIT_LOOPER = _IO('c', 13),
        BC_REQUEST_DEATH_NOTIFICATION = _IOW('c', 14, struct binder_ptr_cookie),
        BC_CLEAR_DEATH_NOTIFICATION = _IOW('c', 15, struct binder_ptr_cookie),
        BC_DEAD_BINDER_DONE = _IOW('c', 16, void *),
};
```

在上述枚举命令成员中，重要的是 BC_TRANSACTION 和 BC_REPLY 命令，被作为发送操作的命令，其数据参数都是 binder_transaction_data 结构体。其中前者用于翻译和解析将要被处理的事件数据，而后者则是事件处理完成之后对返回“结果数据”的操作命令。

5.1.11　枚举 BinderDriverReturnProtocol

在枚举 BinderDriverReturnProtocol 中定义了读操作命令协议，具体实现代码如下所示。

```
enum BinderDriverReturnProtocol {
        BR_ERROR = _IOR('r', 0, int),
        BR_OK = _IO('r', 1),
        /* No parameters! */
```

```
        BR_TRANSACTION = _IOR('r', 2, struct binder_transaction_data),
        BR_REPLY = _IOR('r', 3, struct binder_transaction_data),
        BR_ACQUIRE_RESULT = _IOR('r', 4, int),
        BR_DEAD_REPLY = _IO('r', 5),
        BR_TRANSACTION_COMPLETE = _IO('r', 6),
        BR_INCREFS = _IOR('r', 7, struct binder_ptr_cookie),
        BR_ACQUIRE = _IOR('r', 8, struct binder_ptr_cookie),
        BR_RELEASE = _IOR('r', 9, struct binder_ptr_cookie),
        BR_DECREFS = _IOR('r', 10, struct binder_ptr_cookie),
        BR_ATTEMPT_ACQUIRE = _IOR('r', 11, struct binder_pri_ptr_cookie),
        BR_NOOP = _IO('r', 12),
        BR_SPAWN_LOOPER = _IO('r', 13),
        BR_FINISHED = _IO('r', 14),
        BR_DEAD_BINDER = _IOR('r', 15, void *),
        BR_CLEAR_DEATH_NOTIFICATION_DONE = _IOR('r', 16, void *),
        BR_FAILED_REPLY = _IO('r', 17),
};
```

在上述命令中，BC_TRANSACTION 和 BC_REPLY 命令被作为发送操作命令，其数据参数都是 binder_transaction_data 结构体。其中前者用于翻译和解析将要被处理的事件数据，而后者则是事件处理完成之后对返回“结果数据”的操作命令。

5.1.12 结构体 binder_ptr_cookie 和 binder_transaction_data

binder_ptr_cookie 和 binder_transaction_data 是两个比较重要的结构体，其中 binder_ptr_cookie 表示一个 Binder 实体对象或 Service 组件的死亡接收通知，具体定义代码如下所示。

```
struct binder_ptr_cookie {
        void *ptr;
        void *cookie;
};
```

而 binder_transaction_data 表示在通信过程中传递的数据，具体定义代码如下所示。

```
struct binder_transaction_data {
        union {
                size_t  handle; /* target descriptor of command transaction */
                void    *ptr;   /* target descriptor of return transaction */
        } target;
        void            *cookie;        /* target object cookie */
        unsigned int    code;           /* transaction command */
        /* General information about the transaction. */
        unsigned int    flags;
        pid_t           sender_pid;
        uid_t           sender_euid;
        size_t          data_size;      /* number of bytes of data */
        size_t          offsets_size;   /* number of bytes of offsets */
        union {
                struct {
                        /* transaction data */
                        const void      *buffer;
                        /* offsets from buffer to flat_binder_object structs */
                        const void      *offsets;
```

```
            } ptr;
            uint8_t buf[8];
        } data;
};
```

5.1.13　结构体 flat_binder_object

在 Android 系统中，将在进程之间传递的数据称为 Binder 对象，即 Binder Object。Binder 对象在对应源码中使用结构体 flat_binder_object 来表示，具体代码如下所示。

```
struct flat_binder_object {
        /* 8 bytes for large_flat_header. */
        unsigned long            type;
        unsigned long            flags;
        /* 8 bytes of data. */
        union {
                void             *binder;        /* local object */
                signed long      handle;         /* remote object */
        };
        /* extra data associated with local object */
        void                     *cookie;
};
```

各个成员的具体说明如下。

- ☑ type：描述了 Binder 的类型，传输的数据是一个复用数据联合体。对于 Binder 类型来说，数据是一个 Binder 本地对象。
- ☑ handle：是一个远程的 handle 句柄。假如 A 有个对象 O，对于 A 来说，O 就是一个本地的 Binder 对象；如果 B 想访问 A 的 O 对象，对于 B 来说，O 就是一个 handle，所以 handle 和 Binder 都指向 O。
- ☑ cookie：如果是本地对象，Binder 还可以带有额外的数据，这些数据将被保存到 cookie 字段中。
- ☑ flags：表示传输方式，例如同步和异步等，其值同样使用一个 enum 来表示，具体定义代码如下所示。

```
enum transaction_flags {
        TF_ONE_WAY        = 0x01, /* this is a one-way call: async, no return */
        TF_ROOT_OBJECT    = 0x04, /* contents are the component's root object */
        TF_STATUS_CODE    = 0x08, /* contents are a 32-bit status code */
        TF_ACCEPT_FDS     = 0x10, /* allow replies with file descriptors */
};
```

5.1.14　设备初始化

我们可以在文件 binder.c 中找到该初始化函数 binder_init()，具体定义代码如下所示。

```
static int __init binder_init(void)
{
        int ret;

        binder_deferred_workqueue = create_singlethread_workqueue("binder");
        if (!binder_deferred_workqueue)
                return -ENOMEM;

        binder_debugfs_dir_entry_root = debugfs_create_dir("binder", NULL);
```

```
        if (binder_debugfs_dir_entry_root)
                binder_debugfs_dir_entry_proc = debugfs_create_dir("proc",
                                                binder_debugfs_dir_entry_root);
        ret = misc_register(&binder_miscdev);
        if (binder_debugfs_dir_entry_root) {
                debugfs_create_file("state",
                                        S_IRUGO,
                                        binder_debugfs_dir_entry_root,
                                        NULL,
                                        &binder_state_fops);
                debugfs_create_file("stats",
                                        S_IRUGO,
                                        binder_debugfs_dir_entry_root,
                                        NULL,
                                        &binder_stats_fops);
                debugfs_create_file("transactions",
                                        S_IRUGO,
                                        binder_debugfs_dir_entry_root,
                                        NULL,
                                        &binder_transactions_fops);
                debugfs_create_file("transaction_log",
                                        S_IRUGO,
                                        binder_debugfs_dir_entry_root,
                                        &binder_transaction_log,
                                        &binder_transaction_log_fops);
                debugfs_create_file("failed_transaction_log",
                                        S_IRUGO,
                                        binder_debugfs_dir_entry_root,
                                        &binder_transaction_log_failed,
                                        &binder_transaction_log_fops);
        }
        return ret;
}
```

binder_init()是 Binder 驱动的初始化函数，在实现时需要调用设备驱动接口。Android Binder 设备驱动接口函数是 device_initcall()，使用 module_init 和 module_exit 是为了同时兼容支持静态编译的驱动模块（buildin）和动态编译的驱动模块（module）。Binder 使用 device_initcall 的目的就是不让 Binder 驱动支持动态编译，而且需要在内核（Kernel）做镜像。initcall 用于注册进行初始化的函数，如果的确需要将 Binder 驱动修改为动态的内核模块，可以直接将 device_initcall 修改为 module_init，并增加 module_exit 的驱动卸载接口函数。

在注册 Binder 驱动为 Misc 设备时，指定了 Binder 驱动的 miscdevice，具体实现代码如下所示。

```
static struct miscdevice binder_miscdev = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = "binder",
        .fops = &binder_fops
};
```

Binder 设备的主设备号为 10，此设备号是动态获得的，各个参数的具体说明如下。

- ☑ MISC_DYNAMIC_MINOR：.minor 被设置为动态获得设备号 MISC_DYNAMIC_MINOR。
- ☑ .name：表示设备名称。
- ☑ file_operations：指定了该设备的 file_operations 结构体，定义代码如下所示。

```
static struct file_operations binder_fops = {
	.owner = THIS_MODULE,
	.poll = binder_poll,
	.unlocked_ioctl = binder_ioctl,
	.mmap = binder_mmap,
	.open = binder_open,
	.flush = binder_flush,
	.release = binder_release,
};
```

在 Android 系统中，任何驱动程序都具备向用户空间的程序提供操作接口的功能。这个接口是一个标准接口，Android Binder 驱动提供了操作设备文件（/dev/binder）的接口。正如 binder_fops 所描述的 file_operations 结构体一样，其中主要包括 binder_poll、binder_ioctl、binder_mmap、binder_open、binder_flush、binder_release 等标准操作接口。

5.1.15　打开 Binder 设备文件

在 Android 系统的 Binder 机制中，函数 binder_open()的功能是打开 Binder 设备文件/dev/binder。在 Android 系统中，底层驱动的任何一个进程及线程都可以打开一个 Binder 设备，其打开过程的实现代码如下所示。

```
static int binder_open(struct inode *nodp, struct file *filp)
{
		struct binder_proc *proc;
		binder_debug(BINDER_DEBUG_OPEN_CLOSE, "binder_open: %d:%d\n",
				current->group_leader->pid, current->pid);
		proc = kzalloc(sizeof(*proc), GFP_KERNEL);
		if (proc == NULL)
			return -ENOMEM;
		get_task_struct(current);
		proc->tsk = current;
		INIT_LIST_HEAD(&proc->todo);
		init_waitqueue_head(&proc->wait);
		proc->default_priority = task_nice(current);
		binder_lock(__func__);
		binder_stats_created(BINDER_STAT_PROC);
		hlist_add_head(&proc->proc_node, &binder_procs);
		proc->pid = current->group_leader->pid;
		INIT_LIST_HEAD(&proc->delivered_death);
		filp->private_data = proc;
		binder_unlock(__func__);
		if (binder_debugfs_dir_entry_proc) {
			char strbuf[11];
			snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
			proc->debugfs_entry = debugfs_create_file(strbuf, S_IRUGO,
				binder_debugfs_dir_entry_proc, proc, &binder_proc_fops);
		}
		return 0;
}
```

函数 binder_open 的具体实现流程如下所示。

（1）创建并分配一个 binder_proc 空间来保存 Binder 数据。

（2）增加当前线程/进程的引用计数，给 binder_proc 的 tsk 字段赋值。

（3）实现 binder_proc 队列的初始化，主要包括：

☑ 使用 INIT_LIST_HEAD 初始化链表头 todo。

☑ 使用 init_waitqueue_head 初始化等待队列 wait。

☑ 设置默认优先级（default_priority）为当前进程的 nice 值（通过 task_nice 得到当前进程的 nice 值）。

（4）增加 BINDER_STAT_PROC 的对象计数，并通过 hlist_add_head 把创建的 binder_proc 对象添加到全局的 binder_proc 哈希表中，这样任何一个进程就都可以访问到其他进程的 binder_proc 对象。

（5）把当前进程（或线程）的线程组的 pid（pid 指向线程 id）赋值给 proc 的 pid 字段，同时把创建的 binder_proc 对象指针赋值给 filp 的 private_data 对象并保存起来。

（6）在 binder proc 目录中创建只读文件/proc/binder/proc/$pid，功能是输出当前 binder proc 对象的状态。文件名以 pid 命名，但是该 pid 字段并不是当前进程/线程的 ID，而是线程组的 pid，表示是线程组中第一个线程的 pid（因为上面是将 current->group_leader->pid 赋值给该 pid 字段的），并且在创建该文件时也指定了操作该文件的函数接口为 binder_read_proc_proc，此函数的参数表示创建的 binder_proc 对象 proc。

再看函数 binder_release()，此函数与函数 binder_open()的功能相反。当 Binder 驱动退出时，通过函数 binder_release()来释放在打开以及其他操作过程中分配的空间，并且同时清理相关的数据信息。函数 binder_release 的具体实现代码如下所示。

```
static int binder_release(struct inode *nodp, struct file *filp)
{
	struct binder_proc *proc = filp->private_data;
	debugfs_remove(proc->debugfs_entry);
	binder_defer_work(proc, BINDER_DEFERRED_RELEASE);
	return 0;
}
```

在上述代码中，首先获取使用 private_data 数据的权限，找到当前进程、线程的 pid，这样可以得到在 open 过程中创建的以 pid 命名的用来输出当前 binder proc 对象状态的只读文件；然后调用函数 remove_proc_entry 实现删除操作；最后通过函数 binder_defer_work 和其参数 BINDER_DEFERRED_ RELEASE 释放整个 binder_proc 对象的数据和分配的空间。

5.1.16 实现内存映射

在 Android 系统中，当打开 Binder 设备文件/dev/binder 后，需要调用函数 mmap 把设备内存映射到用户进程地址空间中，这样就可以像操作用户内存那样操作设备内存了。在 Binder 设备中，对内存的映射操作是有限制的，例如 Binder 不能映射具有写权限的内存区域，最大能映射 4MB 的内存区域等。在 Android 系统中，大多数设备本身具有设备映射的设备内存，或者是在驱动初始化时由 vmalloc 或 kmalloc 等内核内存函数分配的，在 mmap 操作时分配 Binder 的设备内存。

函数 mmap 实现分配功能的实现流程如下。

（1）在内核虚拟映射表上获取一个可以使用的区域。

（2）分配物理页，并把物理页映射到获取的虚拟空间上。

（3）每个进程/线程只能执行一次映射操作，后面的操作都会返回错误。

函数 mmap 的具体实现流程如下。

（1）检查内存映射条件，包括映射内存大小（4MB）、flags、是否是第一次 mmap 等。

（2）获得地址空间，并把此空间的地址记录在进程信息（buffer）中。

（3）分配物理页面（pages）并记录下来。

（4）将 buffer 插入到进程信息的 buffer 列表中。

（5）调用函数 binder_update_page_range()将分配的物理页面和 vm 空间对应起来。

（6）调用函数 binder_insert_free_buffer()把进程中的 buffer 插入到进程信息中。

函数 mmap 的具体实现代码如下所示。

```
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
	int ret;
	struct vm_struct *area;
	struct binder_proc *proc = filp->private_data;
	const char *failure_string;
	struct binder_buffer *buffer;

	if ((vma->vm_end - vma->vm_start) > SZ_4M)
		vma->vm_end = vma->vm_start + SZ_4M;

	binder_debug(BINDER_DEBUG_OPEN_CLOSE,
			"binder_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
			proc->pid, vma->vm_start, vma->vm_end,
			(vma->vm_end - vma->vm_start) / SZ_1K, vma->vm_flags,
			(unsigned long)pgprot_val(vma->vm_page_prot));

	if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
		ret = -EPERM;
		failure_string = "bad vm_flags";
		goto err_bad_arg;
	}
	vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;

	mutex_lock(&binder_mmap_lock);
	if (proc->buffer) {
		ret = -EBUSY;
		failure_string = "already mapped";
		goto err_already_mapped;
	}

	area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
	if (area == NULL) {
		ret = -ENOMEM;
		failure_string = "get_vm_area";
		goto err_get_vm_area_failed;
	}
	proc->buffer = area->addr;
	proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;
	mutex_unlock(&binder_mmap_lock);

#ifdef CONFIG_CPU_CACHE_VIPT
	if (cache_is_vipt_aliasing()) {
		while (CACHE_COLOUR((vma->vm_start ^ (uint32_t)proc->buffer))) {
			printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p bad alignment\n",
proc->pid, vma->vm_start, vma->vm_end, proc->buffer);
			vma->vm_start += PAGE_SIZE;
```

```
				}
		}
#endif
		proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma->vm_start) / PAGE_SIZE),
GFP_KERNEL);
		if (proc->pages == NULL) {
				ret = -ENOMEM;
				failure_string = "alloc page array";
				goto err_alloc_pages_failed;
		}
		proc->buffer_size = vma->vm_end - vma->vm_start;

		vma->vm_ops = &binder_vm_ops;
		vma->vm_private_data = proc;

		if (binder_update_page_range(proc, 1, proc->buffer, proc->buffer + PAGE_SIZE, vma)) {
				ret = -ENOMEM;
				failure_string = "alloc small buf";
				goto err_alloc_small_buf_failed;
		}
		buffer = proc->buffer;
		INIT_LIST_HEAD(&proc->buffers);
		list_add(&buffer->entry, &proc->buffers);
		buffer->free = 1;
		binder_insert_free_buffer(proc, buffer);
		proc->free_async_space = proc->buffer_size / 2;
		barrier();
		proc->files = get_files_struct(proc->tsk);
		proc->vma = vma;
		proc->vma_vm_mm = vma->vm_mm;

		/*printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p\n",
				 proc->pid, vma->vm_start, vma->vm_end, proc->buffer);*/
		return 0;

err_alloc_small_buf_failed:
		kfree(proc->pages);
		proc->pages = NULL;
err_alloc_pages_failed:
		mutex_lock(&binder_mmap_lock);
		vfree(proc->buffer);
		proc->buffer = NULL;
err_get_vm_area_failed:
err_already_mapped:
		mutex_unlock(&binder_mmap_lock);
err_bad_arg:
		printk(KERN_ERR "binder_mmap: %d %lx-%lx %s failed %d\n",
				proc->pid, vma->vm_start, vma->vm_end, failure_string, ret);
		return ret;
}
```

在上述代码中，参数 vm_area_struct 是一个结构体，在 mmp 的具体实现中会非常有用。为了优化查找

方法，内核专门维护了 VMA 的链表和树形结构。在结构 vm_area_struct 中，很多成员函数都是用来维护这个树形结构的。VMA 的功能是管理进程地址空间中不同区域的数据结构。该函数首先对内存映射进行检查，主要包括映射内存的大小、flags 以及是否已经映射过了，并判断其映射条件是否合法；然后，通过内核函数 get_vm_area()从系统中申请可用的虚拟内存空间，在内核中申请并保留一块连续的内核虚拟内存空间区域。接着将 binder_proc 的用户地址偏移（即用户进程的 VMA 地址与 Binder 申请的 VMA 地址的偏差）存放到 proc->user_buffer_offset 中；再接着使用 kzalloc()函数根据请求映射的内存空间大小，分配 Binder 的核心数据结构 binder_proc 的 pages 成员，它主要用来保存指向申请的物理页的指针；最后，为 VMA 指定 vm_operations_struct 操作，并且将 vma->vm_private_data 指向核心数据 proc。

到目前为止，就可以真正地开始分配物理内存（page）了。物理内存的分配工作是通过函数 binder_update_page_range()实现的，该函数主要完成如下工作。

☑ alloc_page：分配页面。
☑ map_vm_area：为分配的内存做映射关系。
☑ vm_insert_page：把分配的物理页插入到用户 VMA 区域。

函数 binder_update_page_range()的具体实现代码如下所示。

```
static int binder_update_page_range(struct binder_proc *proc, int allocate,
                                    void *start, void *end,
                                    struct vm_area_struct *vma)
{
        void *page_addr;
        unsigned long user_page_addr;
        struct vm_struct tmp_area;
        struct page **page;
        struct mm_struct *mm;

        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                     "binder: %d: %s pages %p-%p\n", proc->pid,
                     allocate ? "allocate" : "free", start, end);

        if (end <= start)
                return 0;

        trace_binder_update_page_range(proc, allocate, start, end);

        if (vma)
                mm = NULL;
        else
                mm = get_task_mm(proc->tsk);

        if (mm) {
                down_write(&mm->mmap_sem);
                vma = proc->vma;
                if (vma && mm != proc->vma_vm_mm) {
                        pr_err("binder: %d: vma mm and task mm mismatch\n",
                               proc->pid);
                        vma = NULL;
                }
        }
```

```
	if (allocate == 0)
		goto free_range;

	if (vma == NULL) {
		printk(KERN_ERR "binder: %d: binder_alloc_buf failed to "
		       "map pages in userspace, no vma\n", proc->pid);
		goto err_no_vma;
	}

	for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
		int ret;
		struct page **page_array_ptr;
		page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];

		BUG_ON(*page);
		*page = alloc_page(GFP_KERNEL | __GFP_HIGHMEM | __GFP_ZERO);
		if (*page == NULL) {
			printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
			       "for page at %p\n", proc->pid, page_addr);
			goto err_alloc_page_failed;
		}
		tmp_area.addr = page_addr;
		tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
		page_array_ptr = page;
		ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
		if (ret) {
			printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
			       "to map page at %p in kernel\n",
			       proc->pid, page_addr);
			goto err_map_kernel_failed;
		}
		user_page_addr =
			(uintptr_t)page_addr + proc->user_buffer_offset;
		ret = vm_insert_page(vma, user_page_addr, page[0]);
		if (ret) {
			printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
			       "to map page at %lx in userspace\n",
			       proc->pid, user_page_addr);
			goto err_vm_insert_page_failed;
		}
		/* vm_insert_page does not seem to increment the refcount */
	}
	if (mm) {
		up_write(&mm->mmap_sem);
		mmput(mm);
	}
	return 0;

free_range:
	for (page_addr = end - PAGE_SIZE; page_addr >= start;
	     page_addr -= PAGE_SIZE) {
```

```
			page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
			if (vma)
				zap_page_range(vma, (uintptr_t)page_addr +
					proc->user_buffer_offset, PAGE_SIZE, NULL);
err_vm_insert_page_failed:
			unmap_kernel_range((unsigned long)page_addr, PAGE_SIZE);
err_map_kernel_failed:
			__free_page(*page);
			*page = NULL;
err_alloc_page_failed:
			;
		}
err_no_vma:
		if (mm) {
			up_write(&mm->mmap_sem);
			mmput(mm);
		}
		return -ENOMEM;
}
```

其中 vm_operations_struct 只包括了一个打开操作和一个关闭操作，具体的定义代码如下所示。

```
static struct vm_operations_struct binder_vm_ops = {
		.open = binder_vma_open,
		.close = binder_vma_close,
};
```

5.1.17　释放物理页面

在 Android 系统的 Binder 机制中，函数 binder_insert_free_buffer()的功能是进程中的 buffer 插入到进程信息中。也就是说，通过此函数能够将一个空闲内核缓冲区加入到进程中的空闲内核缓冲区的红黑树中。函数 binder_insert_free_buffer()的具体实现代码如下所示。

```
static void binder_insert_free_buffer(struct binder_proc *proc,
						struct binder_buffer *new_buffer)
{
		struct rb_node **p = &proc->free_buffers.rb_node;
		struct rb_node *parent = NULL;
		struct binder_buffer *buffer;
		size_t buffer_size;
		size_t new_buffer_size;
		BUG_ON(!new_buffer->free);
		new_buffer_size = binder_buffer_size(proc, new_buffer);
		binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
				"binder: %d: add free buffer, size %zd, "
				"at %p\n", proc->pid, new_buffer_size, new_buffer);
		while (*p) {
			parent = *p;
			buffer = rb_entry(parent, struct binder_buffer, rb_node);
			BUG_ON(!buffer->free);
			buffer_size = binder_buffer_size(proc, buffer);
			if (new_buffer_size < buffer_size)
```

```
                    p = &parent->rb_left;
            else
                    p = &parent->rb_right;
    }
    rb_link_node(&new_buffer->rb_node, parent, p);
    rb_insert_color(&new_buffer->rb_node, &proc->free_buffers);
}
```

5.1.18　分配内核缓冲区

在 Android 系统中，binder 在使用 buffer 时一次声明一个 proc（对应一个进程）的 buffer 总大小，然后分配一页并做好映射。在使用时如果发现空间不足，会接着映射并把这个 buffer 拆成两个，并把剩余的继续放到 free_buffers 中。在 Binder 驱动程序中，函数*binder_alloc_buf()的功能是分配内核缓冲区，具体代码如下所示。

```
static struct binder_buffer *binder_alloc_buf(struct binder_proc *proc,
                                              size_t data_size,
                                              size_t offsets_size, int is_async)
{
    struct rb_node *n = proc->free_buffers.rb_node;
    struct binder_buffer *buffer;
    size_t buffer_size;
    struct rb_node *best_fit = NULL;
    void *has_page_addr;
    void *end_page_addr;
    size_t size;
    if (proc->vma == NULL) {
            printk(KERN_ERR "binder: %d: binder_alloc_buf, no vma\n",
                   proc->pid);
            return NULL;
    }
    size = ALIGN(data_size, sizeof(void *)) +
            ALIGN(offsets_size, sizeof(void *));
    if (size < data_size || size < offsets_size) {
            binder_user_error("binder: %d: got transaction with invalid "
                    "size %zd-%zd\n", proc->pid, data_size, offsets_size);
            return NULL;
    }
    if (is_async &&
        proc->free_async_space < size + sizeof(struct binder_buffer)) {
            binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                         "binder: %d: binder_alloc_buf size %zd"
                         "failed, no async space left\n", proc->pid, size);
            return NULL;
    }
    while (n) {
            buffer = rb_entry(n, struct binder_buffer, rb_node);
            BUG_ON(!buffer->free);
            buffer_size = binder_buffer_size(proc, buffer);
            if (size < buffer_size) {
                    best_fit = n;
                    n = n->rb_left;
```

```
			} else if (size > buffer_size)
				n = n->rb_right;
			else {
				best_fit = n;
				break;
			}
		}
		if (best_fit == NULL) {
			printk(KERN_ERR "binder: %d: binder_alloc_buf size %zd failed, "
			       "no address space\n", proc->pid, size);
			return NULL;
		}
		if (n == NULL) {
			buffer = rb_entry(best_fit, struct binder_buffer, rb_node);
			buffer_size = binder_buffer_size(proc, buffer);
		}
		binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
			     "binder: %d: binder_alloc_buf size %zd got buff"
			     "er %p size %zd\n", proc->pid, size, buffer, buffer_size);
		has_page_addr =
			(void *)(((uintptr_t)buffer->data + buffer_size) & PAGE_MASK);
		if (n == NULL) {
			if (size + sizeof(struct binder_buffer) + 4 >= buffer_size)
				buffer_size = size; /* no room for other buffers */
			else
				buffer_size = size + sizeof(struct binder_buffer);
		}
		end_page_addr =
			(void *)PAGE_ALIGN((uintptr_t)buffer->data + buffer_size);
		if (end_page_addr > has_page_addr)
			end_page_addr = has_page_addr;
		if (binder_update_page_range(proc, 1,
		    (void *)PAGE_ALIGN((uintptr_t)buffer->data), end_page_addr, NULL))
			return NULL;
		rb_erase(best_fit, &proc->free_buffers);
		buffer->free = 0;
		binder_insert_allocated_buffer(proc, buffer);
		if (buffer_size != size) {
			struct binder_buffer *new_buffer = (void *)buffer->data + size;
			list_add(&new_buffer->entry, &buffer->entry);
			new_buffer->free = 1;
			binder_insert_free_buffer(proc, new_buffer);
		}
		binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
			     "binder: %d: binder_alloc_buf size %zd got "
			     "%p\n", proc->pid, size, buffer);
		buffer->data_size = data_size;
		buffer->offsets_size = offsets_size;
		buffer->async_transaction = is_async;
		if (is_async) {
			proc->free_async_space -= size + sizeof(struct binder_buffer);
			binder_debug(BINDER_DEBUG_BUFFER_ALLOC_ASYNC,
```

```
                                    "binder: %d: binder_alloc_buf size %zd "
                                    "async free %zd\n", proc->pid, size,
                                    proc->free_async_space);
        }
        return buffer;
}
```

再看函数 binder_insert_allocated_buffer()，功能是将分配的内核缓冲区添加到目标进程的已分配物理页面的内核缓冲区红黑树中。函数 binder_insert_allocated_buffer()的具体实现代码如下所示。

```
static void binder_insert_allocated_buffer(struct binder_proc *proc,
                                           struct binder_buffer *new_buffer)
{
        struct rb_node **p = &proc->allocated_buffers.rb_node;
        struct rb_node *parent = NULL;
        struct binder_buffer *buffer;
        BUG_ON(new_buffer->free);
        while (*p) {
                parent = *p;
                buffer = rb_entry(parent, struct binder_buffer, rb_node);
                BUG_ON(buffer->free);
                if (new_buffer < buffer)
                        p = &parent->rb_left;
                else if (new_buffer > buffer)
                        p = &parent->rb_right;
                else
                        BUG();
        }
        rb_link_node(&new_buffer->rb_node, parent, p);
        rb_insert_color(&new_buffer->rb_node, &proc->allocated_buffers);
}
```

5.1.19 释放内核缓冲区

在 Android 系统中，函数 binder_free_buf()的功能是释放内核缓冲区的操作，具体实现代码如下所示。

```
static void binder_free_buf(struct binder_proc *proc,
                            struct binder_buffer *buffer)
{
        size_t size, buffer_size;
        //计算要释放的内核缓冲区 buffer 的大小，保存在 buffer_size 中
        buffer_size = binder_buffer_size(proc, buffer);
        //计算数据缓冲区和偏移数组缓冲区的大小，并保存在 size 中
        size = ALIGN(buffer->data_size, sizeof(void *)) +
                ALIGN(buffer->offsets_size, sizeof(void *));

        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                        "binder: %d: binder_free_buf %p size %zd buffer"
                        "_size %zd\n", proc->pid, buffer, size, buffer_size);

        BUG_ON(buffer->free);
        BUG_ON(size > buffer_size);
        BUG_ON(buffer->transaction != NULL);
        BUG_ON((void *)buffer < proc->buffer);
```

```
		BUG_ON((void *)buffer > proc->buffer + proc->buffer_size);
		//检查要释放的内核缓冲区 buffer 是否用于异步事物
		if (buffer->async_transaction) {
				proc->free_async_space += size + sizeof(struct binder_buffer);

				binder_debug(BINDER_DEBUG_BUFFER_ALLOC_ASYNC,
							"binder: %d: binder_free_buf size %zd "
							"async free %zd\n", proc->pid, size,
							proc->free_async_space);
		}
		//释放内核缓冲区
		binder_update_page_range(proc, 0,
				(void *)PAGE_ALIGN((uintptr_t)buffer->data),
				(void *)(((uintptr_t)buffer->data + buffer_size) & PAGE_MASK),
				NULL);
		rb_erase(&buffer->rb_node, &proc->allocated_buffers);
		buffer->free = 1;
		if (!list_is_last(&buffer->entry, &proc->buffers)) {
				struct binder_buffer *next = list_entry(buffer->entry.next,
										struct binder_buffer, entry);
				if (next->free) {
						rb_erase(&next->rb_node, &proc->free_buffers);
						binder_delete_free_buffer(proc, next);
				}
		}
		if (proc->buffers.next != &buffer->entry) {
				struct binder_buffer *prev = list_entry(buffer->entry.prev,
										struct binder_buffer, entry);
				if (prev->free) {
						binder_delete_free_buffer(proc, buffer);
						rb_erase(&prev->rb_node, &proc->free_buffers);
						buffer = prev;
				}
		}
		binder_insert_free_buffer(proc, buffer);
}
```

再看函数*buffer_start_page()和*buffer_end_page()，用于计算结构体 binder_buffer 所占用的虚拟页面的地址，具体实现代码如下所示。

```
static void *buffer_start_page(struct binder_buffer *buffer)
{
		return (void *)((uintptr_t)buffer & PAGE_MASK);
}
static void *buffer_end_page(struct binder_buffer *buffer)
{
		return (void *)(((uintptr_t)(buffer + 1) - 1) & PAGE_MASK);
}
```

再看函数 binder_delete_free_buffer()，功能是删除结构体 binder_buffer，具体实现代码如下所示。

```
static void binder_delete_free_buffer(struct binder_proc *proc,
							struct binder_buffer *buffer)
{
		struct binder_buffer *prev, *next = NULL;
```

```
		int free_page_end = 1;
		int free_page_start = 1;

		BUG_ON(proc->buffers.next == &buffer->entry);
		prev = list_entry(buffer->entry.prev, struct binder_buffer, entry);
		BUG_ON(!prev->free);
		if (buffer_end_page(prev) == buffer_start_page(buffer)) {
				free_page_start = 0;
				if (buffer_end_page(prev) == buffer_end_page(buffer))
						free_page_end = 0;
				binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
							"binder: %d: merge free, buffer %p "
							"share page with %p\n", proc->pid, buffer, prev);
		}

		if (!list_is_last(&buffer->entry, &proc->buffers)) {
				next = list_entry(buffer->entry.next,
								struct binder_buffer, entry);
				if (buffer_start_page(next) == buffer_end_page(buffer)) {
						free_page_end = 0;
						if (buffer_start_page(next) ==
							buffer_start_page(buffer))
								free_page_start = 0;
						binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
									"binder: %d: merge free, buffer"
									" %p share page with %p\n", proc->pid,
									buffer, prev);
				}
		}
		list_del(&buffer->entry);
		if (free_page_start || free_page_end) {
				binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
							"binder: %d: merge free, buffer %p do "
							"not share page%s%s with with %p or %p\n",
							proc->pid, buffer, free_page_start ? "" : " end",
							free_page_end ? "" : " start", prev, next);
				binder_update_page_range(proc, 0, free_page_start ?
						buffer_start_page(buffer) : buffer_end_page(buffer),
						(free_page_end ? buffer_end_page(buffer) :
						buffer_start_page(buffer)) + PAGE_SIZE, NULL);
		}
}
```

5.1.20 查询内核缓冲区

在 Android 系统中，函数*binder_buffer_lookup()的功能是根据一个用户空间地址查询一个内核缓冲区，具体实现代码如下所示。

```
static struct binder_buffer *binder_buffer_lookup(struct binder_proc *proc,
											void __user *user_ptr)
{
```

```
		struct rb_node *n = proc->allocated_buffers.rb_node;
//对于已经分配的 buffer 空间，以内存地址为索引加入红黑树 allocated_buffers 中
		struct binder_buffer *buffer;
		struct binder_buffer *kern_ptr;

		kern_ptr = user_ptr - proc->user_buffer_offset
				- offsetof(struct binder_buffer, data);
/* 进程 ioctl 传下来的指针并不是 binder_buffer 的地址，而直接是 binder_buffer.data 的地址。user_buffer_offset
用户空间和内核空间，被映射区域起始地址之间的偏移。*/
		while (n) {
				buffer = rb_entry(n, struct binder_buffer, rb_node);
				BUG_ON(buffer->free);

				if (kern_ptr < buffer)
						n = n->rb_left;
				else if (kern_ptr > buffer)
						n = n->rb_right;
				else
						return buffer;
		}
		return NULL;
}
```

5.2　Binder 封装库驱动

在 Android 5.0 系统中，在各个层次都有和 Binder 有关的实现。其中主要的 Binder 库由本地原生代码实现。本节将详细讲解 Binder 封装库的基本知识。

5.2.1　Binder 的 3 层结构

在 Android 系统中，Java 和 C++层都定义了有同样功能的供应用程序使用的 Binder 接口。上述功能是通过调用原生 Binder 库实现的，各个实现层次的具体说明如下。

（1）Binder 驱动部分

驱动部分位于 Binder 结构的最底层（即 Linux 内核层），有关这部分的分析已经在本章前面介绍过了。这部分用于实现 Binder 的设备驱动，主要实现如下功能。

☑　组织 Binder 的服务节点。

☑　调用 Binder 相关的处理线程。

☑　完成实际的 Bainder 传输。

（2）Binder Adapter 层

Binder Adapter 层是对 Binder 驱动的封装，主要功能是操作 Binder 驱动。应用程序无须直接和 Binder 驱动程序关联，关联文件包括 IPCThreadState.cpp、ProcessState.cpp 和 Parcel.cpp 中的一些内容。

Binder 核心库是 Binder 框架的核心实现，主要包括 IBinder、Binder（服务器端）和 BpBinder（客户端）。

（3）顶层

顶层的 Binder 框架和具体的客户端/服务端都分别有 Java 和 C++两种实现方案，主要供应用程序使用，

例如摄像头和多媒体，这部分通过调用 Binder 的核心库来实现。

在文件 frameworks/native/include/binder/IInterface.h 中，分别定义了定义类 IInterface、类模板 BnInterface 和 BpInterface。其中类模板 BnInterface 和 BpInterface 用于实现 Service 组件和 Client 组件，具体定义代码如下所示。

```
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface>      queryLocalInterface(const String16& _descriptor);
    virtual const String16&     getInterfaceDescriptor() const;

protected:
    virtual IBinder*            onAsBinder();
};

// ----------------------------------------------------------------------

template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
                                BpInterface(const sp<IBinder>& remote);

protected:
    virtual IBinder*            onAsBinder();
};
```

在使用这两个模板时，起到了双继承的作用。使用者定义一个接口 INTERFACE，然后使用 BnInterface 和 BpInterface 两个模板结合自己的接口，构建自己的 BnXXX 和 BpXXX 两个类。

5.2.2 Binder 驱动的同事——类 BBinder

类模板 BnInterface 继承于类 BBinder，BBinder 是服务的载体，和 Binder 驱动共同工作，保证客户的请求最终是对一个 Binder 对象（BBinder 类）的调用。从 Binder 驱动的角度，每一个服务就是一个 BBinder 类，Binder 驱动负责找出服务对应的 BBinder 类。然后把这个 BBinder 类返回给 IPCThreadState，IPCThreadState 调用 BBinder 的 transact()。BBinder 的 transact()又会调用 onTransact()。BBinder::onTransact()是虚函数，所以实际是调用 BnXXXService::onTransact()，这样就可在 BnXXXService::onTransact()中完成具体的服务函数的调用。整个 BnXXXService 的类关系图如图 5-1 所示。

由图 5-1 可以看出，BnXXXService 包含如下两部分。

☑ IXXXService：服务主体的接口。

☑ BBinder：服务的载体，和 Binder 驱动共同工作，保证客户的请求最终是对一个 Binder 对象（BBinder 类）的调用。

每一个服务就是一个 BBinder 类，Binder 驱动负责找出服务对应的 BBinder 类。然后把这个 BBinder 类返回给 IPCThreadState，IPCThreadState 调用 BBinder 的 transact()。BBinder 的 transact()又会调用 onTransact()。BBinder::onTransact()是虚函数，所以实际是调用 BnXXXService::onTransact()，这样就可在 BnXXXService::onTransact()中完成具体的服务函数的调用。

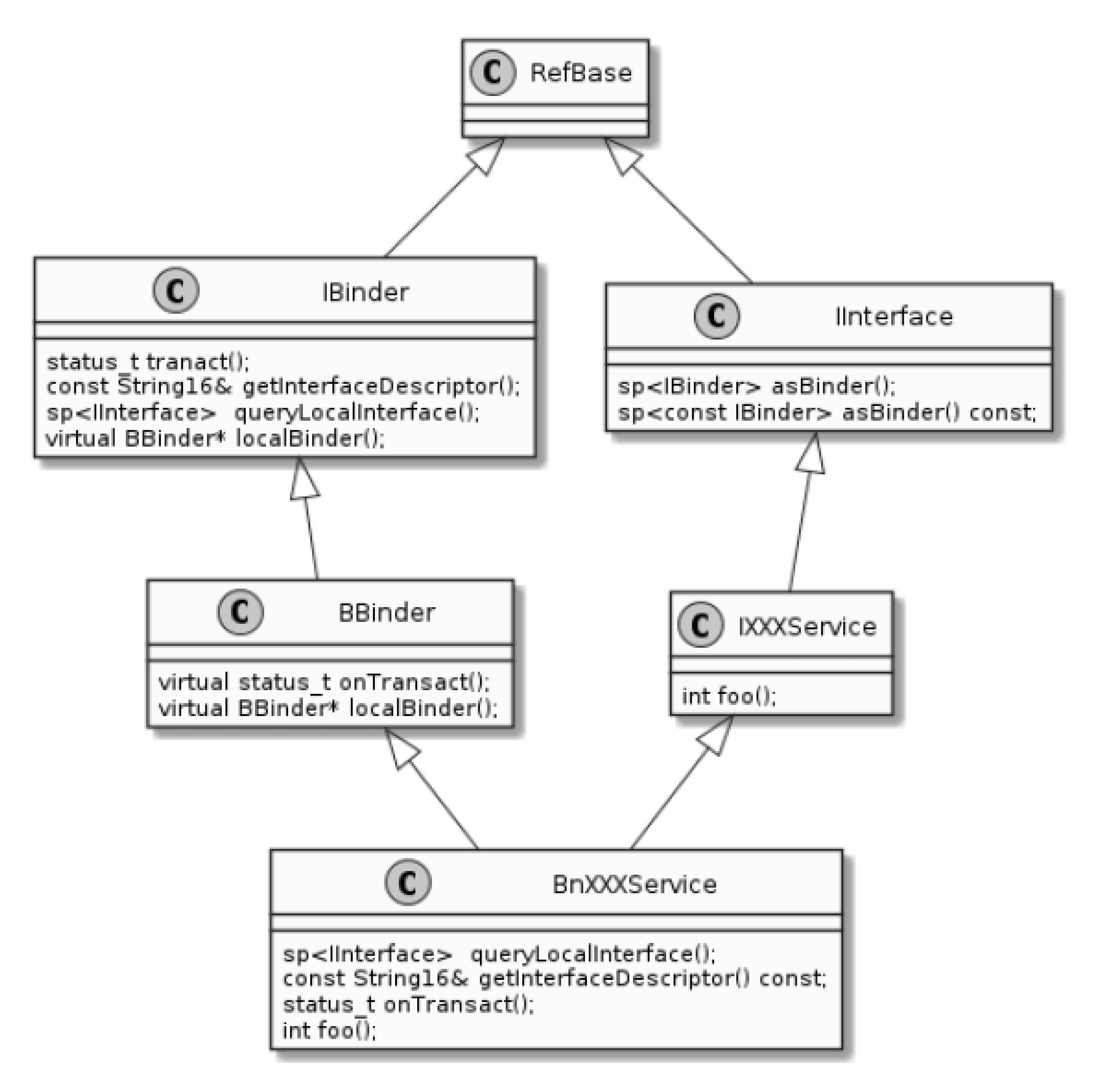


图 5-1　类关系图

在文件 frameworks/native/include/binder/Binder.h 中，定义类 BBinder 的代码如下所示。

```
class BBinder : public IBinder
{
public:
                                BBinder();
    virtual const String16& getInterfaceDescriptor() const;
    virtual bool            isBinderAlive() const;
    virtual status_t        pingBinder();
    virtual status_t        dump(int fd, const Vector<String16>& args);
    virtual status_t        transact(   uint32_t code,
                                        const Parcel& data,
                                        Parcel* reply,
                                        uint32_t flags = 0);
    virtual status_t        linkToDeath(const sp<DeathRecipient>& recipient,
                                        void* cookie = NULL,
                                        uint32_t flags = 0);
    virtual status_t        unlinkToDeath(  const wp<DeathRecipient>& recipient,
                                            void* cookie = NULL,
                                            uint32_t flags = 0,
                                            wp<DeathRecipient>* outRecipient = NULL);
    virtual void            attachObject(   const void* objectID,
                                            void* object,
                                            void* cleanupCookie,
                                            object_cleanup_func func);
    virtual void*           findObject(const void* objectID) const;
    virtual void            detachObject(const void* objectID);
    virtual BBinder*        localBinder();
protected:
    virtual                 ~BBinder();
    virtual status_t        onTransact( uint32_t code,
```

```
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);
private:
                        BBinder(const BBinder& o);
            BBinder&     operator=(const BBinder& o);
    class Extras;
            Extras*      mExtras;
            void*        mReserved0;
};
```

在类 BBinder 中，当一个 Binder 代理对象通过 Binder 驱动程序向一个 Binder 本地对象发出一个进程通信请求时，Binder 驱动程序会调用该 Binder 本地对象的成员函数 transact()来处理这个请求。函数 transact()在文件 frameworks/native/libs/binder/Binder.cpp 中实现，具体代码如下所示。

```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    data.setDataPosition(0);

    status_t err = NO_ERROR;
    switch (code) {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
            err = onTransact(code, data, reply, flags);
            break;
    }
    if (reply != NULL) {
        reply->setDataPosition(0);
    }
    return err;
}
```

在上述代码中，PING_TRANSACTION 请求用来检查对象是否还存在，此处只是简单地把 pingBinder 的返回值返回给调用者，将其他的请求交给 onTransact 来处理。onTransact 是在 Bbinder 中声明的一个 protected 类型的虚函数，此功能在它的子类中实现。

再看另外一个重要的成员函数 onTransact()，功能是分发和业务相关的进程间通信请求。函数 onTransact()也是在文件 frameworks/native/libs/binder/Binder.cpp 中定义，具体实现代码如下所示。

```
status_t BBinder::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch (code) {
        case INTERFACE_TRANSACTION:
            reply->writeString16(getInterfaceDescriptor());
            return NO_ERROR;

        case DUMP_TRANSACTION: {
            int fd = data.readFileDescriptor();
            int argc = data.readInt32();
            Vector<String16> args;
```

```
            for (int i = 0; i < argc && data.dataAvail() > 0; i++) {
                args.add(data.readString16());
            }
            return dump(fd, args);
        }
        case SYSPROPS_TRANSACTION: {
            report_sysprop_change();
            return NO_ERROR;
        }
        default:
            return UNKNOWN_TRANSACTION;
    }
}
```

5.2.3　BpRefBase 代理类

类模板 BpInterface 继承于类 BpRefBase，起到一个代理作用。BpRefBase 以上是业务逻辑（要实现什么功能），BpRefBase 以下是数据传输（通过 Binder 如何将功能实现）。在文件 frameworks/native/include/binder/Binder.h 中，定义类 BpRefBase 的代码如下所示。

```
class BpRefBase : public virtual RefBase
{
protected:
                            BpRefBase(const sp<IBinder>& o);
    virtual                 ~BpRefBase();
    virtual void            onFirstRef();
    virtual void            onLastStrongRef(const void* id);
    virtual bool            onIncStrongAttempted(uint32_t flags, const void* id);

    inline  IBinder*        remote()                { return mRemote; }
    inline  IBinder*        remote() const          { return mRemote; }

private:
                            BpRefBase(const BpRefBase& o);
    BpRefBase&              operator=(const BpRefBase& o);

    IBinder* const          mRemote;
    RefBase::weakref_type*  mRefs;
    volatile int32_t        mState;
};

}; // namespace android
```

类 BpRefBase 中的成员函数 transact()用于向运行在 Server 进程中的 Service 组件发送进程之间的通信请求，这是通过 Binder 驱动程序间接实现的。函数 transact()的具体实现代码如下所示。

```
status_t BpBinder::transact(
uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
// Once a binder has died, it will never come back to life
if (mAlive) {
status_t status = IPCThreadState::self()->transact(
```

```
mHandle, code, data, reply, flags);
if (status == DEAD_OBJECT) mAlive = 0;
return status;
}
return DEAD_OBJECT;
}
```

各个参数的具体说明如下。

☑ code：表示请求的 ID 号。

☑ data：表示请求的参数。

☑ reply：表示返回的结果。

☑ flags：是一些额外的标识，例如 FLAG_ONEWAY，通常为 0。

5.2.4 驱动交互类 IPCThreadState

类 BBinder 和类 BpRefBase 都是通过类 IPCThreadState 和 Binder 的驱动程序交互实现的。类 IPCThreadState 在文件 frameworks/native/include/binder/IPCThreadState.h 中实现，具体实现代码如下所示。

```
class IPCThreadState
{
public:
    static  IPCThreadState*      self();
    static  IPCThreadState*      selfOrNull();   // self(), but won't instantiate
            sp<ProcessState>      process();
            status_t              clearLastError();
            int                   getCallingPid();
            int                   getCallingUid();
            void                  setStrictModePolicy(int32_t policy);
            int32_t               getStrictModePolicy() const;
            void                  setLastTransactionBinderFlags(int32_t flags);
            int32_t               getLastTransactionBinderFlags() const;
            int64_t               clearCallingIdentity();
            void                  restoreCallingIdentity(int64_t token);
            void                  flushCommands();
            void                  joinThreadPool(bool isMain = true);
            // Stop the local process
            void                  stopProcess(bool immediate = true);
            status_t              transact(int32_t handle,
                                           uint32_t code, const Parcel& data,
                                           Parcel* reply, uint32_t flags);
            void                  incStrongHandle(int32_t handle);
            void                  decStrongHandle(int32_t handle);
            void                  incWeakHandle(int32_t handle);
            void                  decWeakHandle(int32_t handle);
            status_t              attemptIncStrongHandle(int32_t handle);
    static  void                  expungeHandle(int32_t handle, IBinder* binder);
            status_t              requestDeathNotification(    int32_t handle,
                                                              BpBinder* proxy);
            status_t              clearDeathNotification( int32_t handle,
                                                          BpBinder* proxy);
    static  void                  shutdown();
```

```
    static  void                    disableBackgroundScheduling(bool disable);
private:
                                    IPCThreadState();
                                    ~IPCThreadState();
            status_t                sendReply(const Parcel& reply, uint32_t flags);
            status_t                waitForResponse(Parcel *reply,
                                                    status_t *acquireResult=NULL);
            status_t                talkWithDriver(bool doReceive=true);
            status_t                writeTransactionData(int32_t cmd,
                                                         uint32_t binderFlags,
                                                         int32_t handle,
                                                         uint32_t code,
                                                         const Parcel& data,
                                                         status_t* statusBuffer);
            status_t                executeCommand(int32_t command);
            void                    clearCaller();
    static  void                    threadDestructor(void *st);
    static  void                    freeBuffer(Parcel* parcel,
                                               const uint8_t* data, size_t dataSize,
                                               const size_t* objects, size_t objectsSize,
                                               void* cookie);
    const   sp<ProcessState>        mProcess;
    const   pid_t                   mMyThreadId;
            Vector<BBinder*>        mPendingStrongDerefs;
            Vector<RefBase::weakref_type*> mPendingWeakDerefs;
            Parcel                  mIn;
            Parcel                  mOut;
            status_t                mLastError;
            pid_t                   mCallingPid;
            uid_t                   mCallingUid;
            int32_t                 mStrictModePolicy;
            int32_t                 mLastTransactionBinderFlags;
};
}; // namespace android
```

在类 IPCThreadState 中，成员函数用于实现数据处理。在 transact 请求中将请求的数据经过 Binder 设备发送给了 Service，Service 处理完请求后，又将结果原路返回给客户端。函数 transact()在文件 frameworks/native/libs/binder/IPCThreadState.cpp 中定义，具体实现代码如下所示。

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err = data.errorCheck();
    flags |= TF_ACCEPT_FDS;
    IF_LOG_TRANSACTIONS() {
        TextOutput::Bundle _b(alog);
        alog << "BC_TRANSACTION thr " << (void*)pthread_self() << " / hand "
            << handle << " / code " << TypeCode(code) << ": "
            << indent << data << dedent << endl;
    }
```

```
    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

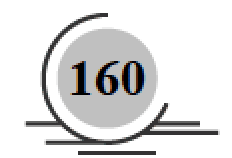
    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        #if 0
        if (code == 4) { // relayout
            ALOGI(">>>>>> CALLING transaction 4");
        } else {
            ALOGI(">>>>>> CALLING transaction %d", code);
        }
        #endif
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        #if 0
        if (code == 4) { // relayout
            ALOGI("<<<<<< RETURNING transaction 4");
        } else {
            ALOGI("<<<<<< RETURNING transaction %d", code);
        }
        #endif

        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_REPLY thr " << (void*)pthread_self() << " / hand "
                << handle << ": ";
            if (reply) alog << indent << *reply << dedent << endl;
            else alog << "(none requested)" << endl;
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }
    return err;
}
```

5.3　初始化 Java 层 Binder 框架

虽然 Java 层 Binder 系统是 Native 层 Binder 系统的一个 Mirror，但是 Mirror 最终还需借助 Native 层的



Binder 系统来开展工作，即它和 Native 层 Binder 有着千丝万缕的关系。正因为如此，所以一定要在 Java 层 Binder 正式工作之前建立这种关系。本节将详细讲解 Java 层 Binder 框架的初始化过程。

5.3.1　搭建交互关系

在 Android 系统中，其中函数 register_android_os_Binder 专门负责搭建 Java Binder 和 Native Binder 的交互关系。此函数在/frameworks/base/core/jni/android_util_Binder.cpp 文件中实现。

函数 register_android_os_Binder()的具体实现代码如下所示。

```
int register_android_os_Binder(JNIEnv* env)
{
  //初始化 Java Binder 类和 Native 层的关系
  if (int_register_android_os_Binder(env) < 0)
        return -1;
  //初始化 Java BinderInternal 类和 Native 层的关系
  if (int_register_android_os_BinderInternal(env) < 0)
        return -1;
 //初始化 Java BinderProxy 类和 Native 层的关系
  if (int_register_android_os_BinderProxy(env) < 0)
        return -1;
  //初始化 Java Parcel 类和 Native 层的关系
   if (int_register_android_os_Parcel(env) < 0)
      return -1;
  return 0;
}
```

根据上面的代码可知，函数 register_android_os_Binder()完成了 Java 层 Binder 架构中最重要的 4 个类的初始化工作。下面将详细分析 Binder 类的初始化进程。

5.3.2　实现 Binder 类的初始化

函数 int_register_android_os_Binder()实现了 Binder 类的初始化工作，此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示。

```
static int int_register_android_os_Binder(JNIEnv* env)
{
jclass clazz;
//kBinderPathName 为 Java 层中 Binder 类的全路径名，  “android/os/Binder”
clazz = env->FindClass(kBinderPathName);
/*
gBinderOffsets 是一个静态类对象，它专门保存 Binder 类的一些在 JNI 层中使用的信息，
如成员函数 execTransact 的 methodID，Binder 类中成员 mObject 的 fieldID

*/
  gBinderOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
  gBinderOffsets.mExecTransact
                          = env->GetMethodID(clazz, "execTransact", "(IIII)Z");
  gBinderOffsets.mObject
                          = env->GetFieldID(clazz, "mObject", "I");
 //注册 Binder 类中 native 函数的实现
  return AndroidRuntime::registerNativeMethods(
```

```
                        env, kBinderPathName,
                        gBinderMethods, NELEM(gBinderMethods));
}
```

从上面的代码可知，gBinderOffsets 对象保存了和 Binder 类相关的某些在 JNI 层中使用的信息。

下一个初始化的类是 BinderInternal，其代码在 int_register_android_os_BinderInternal()函数中。此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示。

```
static int int_register_android_os_BinderInternal(JNIEnv* env)
{
 jclass clazz;
 //根据 BinderInternal 的全路径名找到代表该类的 jclass 对象。全路径名为
 // "com/android/internal/os/BinderInternal"
 clazz = env->FindClass(kBinderInternalPathName);
 //gBinderInternalOffsets 也是一个静态对象，用来保存 BinderInternal 类的一些信息
 gBinderInternalOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
 //获取 forceBinderGc 的 methodID
 gBinderInternalOffsets.mForceGc
                  = env->GetStaticMethodID(clazz, "forceBinderGc", "()V");
  //注册 BinderInternal 类中 native 函数的实现
  return AndroidRuntime::registerNativeMethods(
                        env, kBinderInternalPathName,
                        gBinderInternalMethods, NELEM(gBinderInternalMethods));
}
```

由此可见，int_register_android_os_BinderInternal 的功能和 int_register_android_os_Binder 的功能类似，主要包括以下两个方面。

- ☑ 获取一些有用的 methodID 和 fieldID。这表明 JNI 层一定会向上调用 Java 层的函数。
- ☑ 注册相关类中 native 函数的实现。

5.3.3 实现 BinderProxy 类的初始化

函数 int_register_android_os_BinderProxy()完成了 BinderProxy 类的初始化工作，此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示。

```
static int int_register_android_os_BinderProxy(JNIEnv* env)
{
  jclass clazz;

 clazz = env->FindClass("java/lang/ref/WeakReference");
 //gWeakReferenceOffsets 用来和 WeakReference 类打交道
 gWeakReferenceOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
 //获取 WeakReference 类 get 函数的 methodID
  gWeakReferenceOffsets.mGet= env->GetMethodID(clazz, "get",
                              "()Ljava/lang/Object;");
  clazz = env->FindClass("java/lang/Error");
  //gErrorOffsets 用来和 Error 类打交道
  gErrorOffsets.mClass = (jclass) env->NewGlobalRef(clazz);

  clazz = env->FindClass(kBinderProxyPathName);
  //gBinderProxyOffsets 用来和 BinderProxy 类打交道
  gBinderProxyOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
  gBinderProxyOffsets.mConstructor= env->GetMethodID(clazz, "<init>", "()V");
```

```
    ... //获取 BinderProxy 的一些信息
    clazz = env->FindClass("java/lang/Class");
    //gClassOffsets 用来和 Class 类打交道
    gClassOffsets.mGetName =env->GetMethodID(clazz,
                                    "getName", "()Ljava/lang/String;");
    //注册 BinderProxy native 函数的实现
    return AndroidRuntime::registerNativeMethods(env,
            kBinderProxyPathName,gBinderProxyMethods,
                                    NELEM(gBinderProxyMethods));
}
```

根据上面的代码可知，int_register_android_os_BinderProxy()函数除了初始化 BinderProxy 类外，还获取了 WeakReference 类和 Error 类的一些信息。由此可见，BinderProxy 对象的生命周期会委托 WeakReference 来管理，因此 JNI 层会获取该类 get 函数的 MethodID。

到此为止，Java Binder 几个重要成员的初始化已完成，同时在代码中定义了几个全局静态对象，分别是 gBinderOffsets、gBinderInternalOffsets 和 gBinderProxyOffsets。

5.4　实体对象 binder_node 的驱动

在 Android 系统中，binder_node 用来描述一个 Binder 实体对象。在 Binder 驱动程序中，每个 Service 组件都对应有一个用来描述它在内核中状态的 Binder 实体对象。Android 系统的 Binder 通信框架如图 5-2 所示。

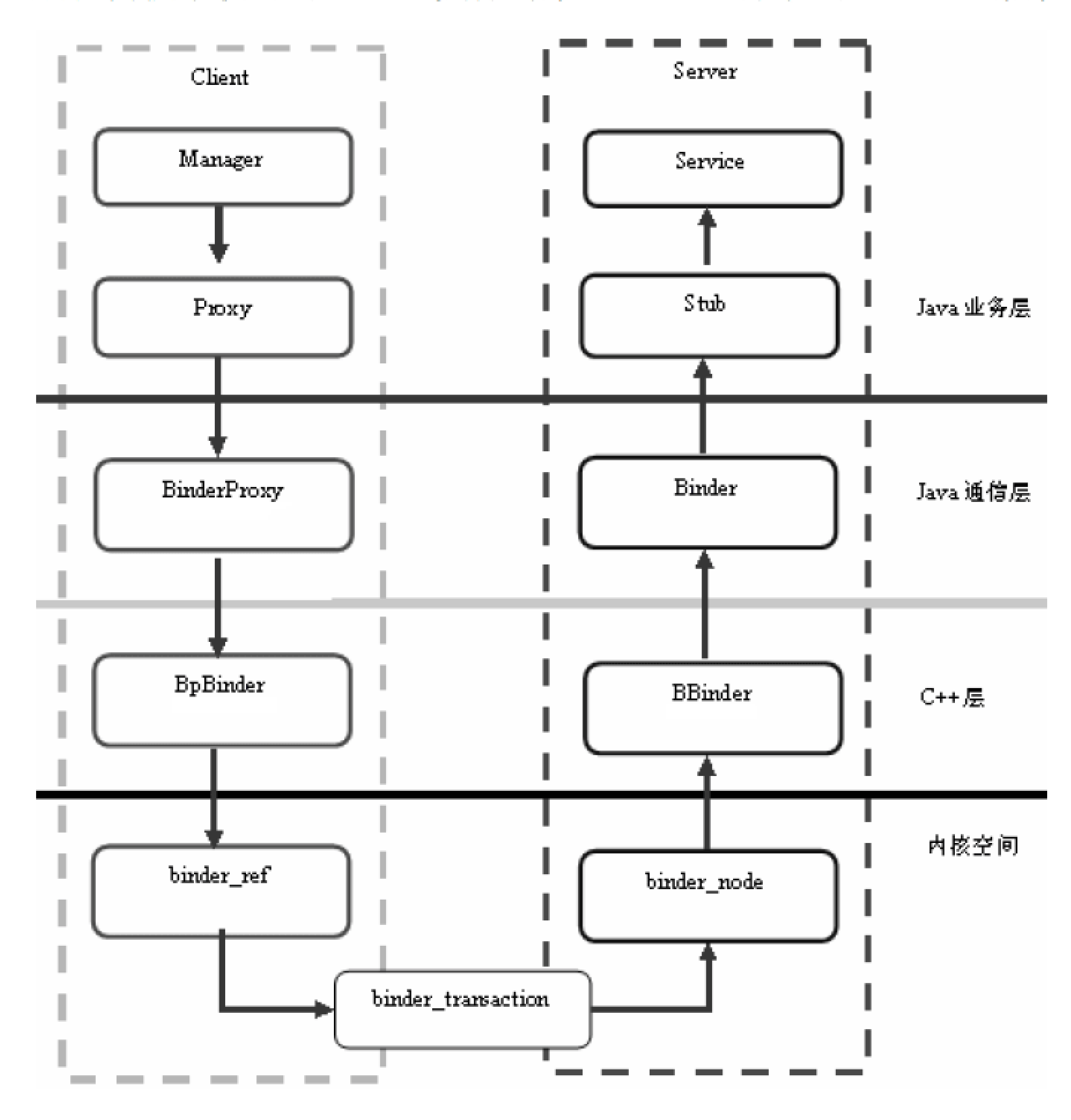


图 5-2　Binder 通信框架图

本节将详细讲解 Android 5.0 实体对象 binder_node 驱动的知识。

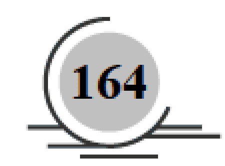
5.4.1　定义实体对象

Binder 实体对象 binder_node 的定义代码如下所示。

```
struct binder_node {
	//调试 ID
	int debug_id;
	//描述一个待处理的工作项
	struct binder_work work;
	union {
		//挂载到宿主进程 binder_proc 的成员变量 nodes 红黑树的节点
		struct rb_node rb_node;
		//当宿主进程死亡，该 binder 实体对象将挂载到全局 binder_dead_nodes 链表中
		struct hlist_node dead_node;
	};
	//指向该 binder 线程的宿主进程
	struct binder_proc *proc;
	//保存所有引用该 binder 实体对象的 binder 引用对象
	struct hlist_head refs;
	//binder 实体对象的强引用计数
	int internal_strong_refs;
	int local_strong_refs;
	unsigned has_strong_ref:1;
	unsigned pending_strong_ref:1;
	unsigned has_weak_ref:1;
	unsigned pending_weak_ref:1;
	//binder 实体对象的弱引用计数
	int local_weak_refs;
	//指向用户空间 service 组件内部的引用计数对象 wekref_impl 的地址
	void __user *ptr;
	//保存用户空间的 service 组件地址
	void __user *cookie;
	//标示该 binder 实体对象是否正在处理一个异步事务
	unsigned has_async_transaction:1;
	//设置该 binder 实体对象是否可以接收包含有文件描述符的 IPC 数据
	unsigned accept_fds:1;
	//binder 实体对象要求处理线程应具备的最小线程优先级
	unsigned min_priority:8;
	//异步事务队列
	struct list_head async_todo;
};
```

通过上述代码可知，在 Binder 驱动中，用户空间中的每一个 Binder 本地对象都对应有一个 Binder 实体对象。各个成员的具体说明如下。

- ☑ proc：指向 Binder 实体对象的宿主进程，宿主进程使用红黑树来维护它内部的所有 Binder 实体对象。
- ☑ rb_node：用来挂载到宿主进程 proc 的 Binder 实体对象红黑树中的节点。
- ☑ dead_node：如果该 Binder 实体对象的宿主进程已经死亡，该 Binder 实体就通过成员变量 dead_node 保存到全局链表 binder_dead_nodes。
- ☑ refs：一个 Binder 实体对象可以被多个 client 引用，成员变量 refs 用来保存所有引用该 Binder 实体



的 Binder 引用对象。

- ☑ internal_strong_refs 和 local_strong_refs：都是用来描述 Binder 实体对象的强引用计数。
- ☑ local_weak_refs：用来描述 Binder 实体对象的弱引用计数。
- ☑ ptr 和 cookie：分别指向用户空间地址，cookie 指向 BBinder 的地址，ptr 指向 BBinder 对象的引用计数地址。
- ☑ has_async_transaction：描述一个 Binder 实体对象是否正在处理一个异步事务，当 Binder 驱动指定某个线程来处理某一事务时，首先将该事务保存到指定线程的 todo 队列中，表示要由该线程来处理该事务。如果是异步事务，Binder 驱动程序就会将它保存在目标 Binder 实体对象的一个异步事务队列 async_todo 中。
- ☑ min_priority：表示一个 Binder 实体对象在处理来自 client 进程请求时，要求处理线程的最小线程优先级。

5.4.2　增加引用计数

在 Binder 驱动程序中，使用函数 binder_inc_node()来增加一个 Binder 实体对象的引用计数。函数 binder_inc_node()在文件/drivers/staging/android/binder.c 中定义，具体实现代码如下所示。

```
static int binder_inc_node(struct binder_node *node, int strong, int internal,
			   struct list_head *target_list)
{
	if (strong) {
		if (internal) {
			if (target_list == NULL &&
			    node->internal_strong_refs == 0 &&
			    !(node == binder_context_mgr_node &&
			    node->has_strong_ref)) {
				printk(KERN_ERR "binder: invalid inc strong "
					"node for %d\n", node->debug_id);
				return -EINVAL;
			}
			node->internal_strong_refs++;
		} else
			node->local_strong_refs++;
		if (!node->has_strong_ref && target_list) {
			list_del_init(&node->work.entry);
			list_add_tail(&node->work.entry, target_list);
		}
	} else {
		if (!internal)
			node->local_weak_refs++;
		if (!node->has_weak_ref && list_empty(&node->work.entry)) {
			if (target_list == NULL) {
				printk(KERN_ERR "binder: invalid inc weak node "
					"for %d\n", node->debug_id);
				return -EINVAL;
			}
			list_add_tail(&node->work.entry, target_list);
		}
```

```
        }
        return 0;
}
```

各个参数的具体说明如下。

- ☑ node：表示要增加引用计数的 Binder 实体对象。
- ☑ strong：表示要增加强引用计数还是要增加弱引用计数。
- ☑ internal：用于区分增加的是内部引用计数还是外部引用计数。
- ☑ target_list：用于指向一个目标进程或目标线程的 todo 队列，当不是 null 值时，表示增加了 Binder 实体对象的引用计数后，需要对应增加它所引用的 Binder 本地对象的引用计数。

5.4.3 减少引用计数

在 Binder 驱动程序中，使用函数 binder_dec_node()来减少一个 Binder 实体对象的引用计数。函数 binder_dec_node()会减少 internal_strong_refs、local_strong_refs 或 local_weak_refs 的使用计数，并删除节点的 work.entry 链表。函数 binder_dec_node()也是在文件/drivers/staging/android/binder.c 中定义，具体实现代码如下所示。

```
static int binder_dec_node(struct binder_node *node, int strong, int internal)
{
        if (strong) {
                if (internal)
                        node->internal_strong_refs--;
                else
                        node->local_strong_refs--;
                if (node->local_strong_refs || node->internal_strong_refs)
                        return 0;
        } else {
                if (!internal)
                        node->local_weak_refs--;
                if (node->local_weak_refs || !hlist_empty(&node->refs))
                        return 0;
        }
        if (node->proc && (node->has_strong_ref || node->has_weak_ref)) {
                if (list_empty(&node->work.entry)) {
                        list_add_tail(&node->work.entry, &node->proc->todo);
                        wake_up_interruptible(&node->proc->wait);
                }
        } else {
                if (hlist_empty(&node->refs) && !node->local_strong_refs &&
                    !node->local_weak_refs) {
                        list_del_init(&node->work.entry);
                        if (node->proc) {
                                rb_erase(&node->rb_node, &node->proc->nodes);
                                binder_debug(BINDER_DEBUG_INTERNAL_REFS,
                                             "binder: refless node %d deleted\n",
                                             node->debug_id);
                        } else {
                                hlist_del(&node->dead_node);
                                binder_debug(BINDER_DEBUG_INTERNAL_REFS,
```

```
                                                   "binder: dead node %d deleted\n",
                                                   node->debug_id);
                                }
                                kfree(node);
                                binder_stats_deleted(BINDER_STAT_NODE);
                        }
                }
                return 0;
}
```

5.5　本地对象 BBinder 驱动

因为 Binder 的功能就是在本地执行其他进程的功能，所以 Binder 机制也是 Android 的一种远程过程调用（RPC）机制。当进程通过 Binder 获取将要调用的进程服务时，不但可以是一个本地对象，而且也可以是一个远程服务的引用。也就是说，Binder 不但可以与本地进程通信，而且还可以与远程进程通信。此处的本地进程就是本节所讲解的本地对象，而远程进程就是远程服务的一个引用。

5.5.1　引用运行的本地对象

在 Android 系统中，Binder 驱动程序通过函数 binder_thread_read()引用运行在 Server 进程中的 Binder 本地对象，此函数在文件 drivers/staging/android/binder.c 中定义。当 service_manager 运行时此函数会一直等待，直到有请求到达为止。函数 binder_thread_read()的具体实现代码如下所示。

```
static int binder_thread_read(struct binder_proc *proc,
                              struct binder_thread *thread,
                              void  __user *buffer, int size,
                              signed long *consumed, int non_block)
{
        void __user *ptr = buffer + *consumed;
        void __user *end = buffer + size;
        int ret = 0;
        int wait_for_proc_work;
        if (*consumed == 0) {
                if (put_user(BR_NOOP, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
        }
retry:
        wait_for_proc_work = thread->transaction_stack == NULL &&
                                list_empty(&thread->todo);
        if (thread->return_error != BR_OK && ptr < end) {
                if (thread->return_error2 != BR_OK) {
                        if (put_user(thread->return_error2, (uint32_t __user *)ptr))
                                return -EFAULT;
                        ptr += sizeof(uint32_t);
                        binder_stat_br(proc, thread, thread->return_error2);
                        if (ptr == end)
```

```
                    goto done;
                thread->return_error2 = BR_OK;
            }
            if (put_user(thread->return_error, (uint32_t __user *)ptr))
                return -EFAULT;
            ptr += sizeof(uint32_t);
            binder_stat_br(proc, thread, thread->return_error);
            thread->return_error = BR_OK;
            goto done;
        }
        thread->looper |= BINDER_LOOPER_STATE_WAITING;
        if (wait_for_proc_work)
            proc->ready_threads++;
        binder_unlock(__func__);
        trace_binder_wait_for_work(wait_for_proc_work,
                                   !!thread->transaction_stack,
                                   !list_empty(&thread->todo));
        if (wait_for_proc_work) {
            if (!(thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
                                    BINDER_LOOPER_STATE_ENTERED))) {
                binder_user_error("binder: %d:%d ERROR: Thread waiting "
                    "for process work before calling BC_REGISTER_"
                    "LOOPER or BC_ENTER_LOOPER (state %x)\n",
                    proc->pid, thread->pid, thread->looper);
                wait_event_interruptible(binder_user_error_wait,
                                         binder_stop_on_user_error < 2);
            }
            binder_set_nice(proc->default_priority);
            if (non_block) {
                if (!binder_has_proc_work(proc, thread))
                    ret = -EAGAIN;
            } else
                ret = wait_event_freezable_exclusive(proc->wait, binder_has_proc_work(proc, thread));
        } else {
            if (non_block) {
                if (!binder_has_thread_work(thread))
                    ret = -EAGAIN;
            } else
                ret = wait_event_freezable(thread->wait, binder_has_thread_work(thread));
        }
        binder_lock(__func__);
        if (wait_for_proc_work)
            proc->ready_threads--;
        thread->looper &= ~BINDER_LOOPER_STATE_WAITING;
        if (ret)
            return ret;
        while (1) {
            uint32_t cmd;
//将用户传进来的 transact 参数复制在本地变量 struct binder_transaction_data tr 中
            struct binder_transaction_data tr;
            struct binder_work *w;
```

```
                struct binder_transaction *t = NULL;
//由于 thread->todo 不为空，执行下列语句
                if (!list_empty(&thread->todo))
                        w = list_first_entry(&thread->todo, struct binder_work, entry);
                else if (!list_empty(&proc->todo) && wait_for_proc_work)
//Service Manager 被唤醒之后，就进入 while 循环开始处理事务了
//此处 wait_for_proc_work 等于 1，并且 proc->todo 不为空，所以从 proc->todo 列表中得到第一个工作项
                        w = list_first_entry(&proc->todo, struct binder_work, entry);
                else {
                        if (ptr - buffer == 4 && !(thread->looper & BINDER_LOOPER_STATE_NEED_RETURN))
/* no data added */
                                goto retry;
                        break;
                }
                if (end - ptr < sizeof(tr) + 4)
                        break;
                switch (w->type) {
//函数调用 binder_transaction 进一步处理
                case BINDER_WORK_TRANSACTION: {
//因为这个工作项的类型为 BINDER_WORK_TRANSACTION，所以通过下面语句得到事务项
                        t = container_of(w, struct binder_transaction, work);
                } break;
//因为 w->type 为 BINDER_WORK_TRANSACTION_COMPLETE
//这是在上面的 binder_transaction 函数设置的，于是执行下面的代码
                case BINDER_WORK_TRANSACTION_COMPLETE: {
                        cmd = BR_TRANSACTION_COMPLETE;
                        if (put_user(cmd, (uint32_t __user *)ptr))
                                return -EFAULT;
                        ptr += sizeof(uint32_t);
                        binder_stat_br(proc, thread, cmd);
                        binder_debug(BINDER_DEBUG_TRANSACTION_COMPLETE,
                                     "binder: %d:%d BR_TRANSACTION_COMPLETE\n",
                                     proc->pid, thread->pid);
//将 w 从 thread->todo 删除
                        list_del(&w->entry);
                        kfree(w);
                        binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
                } break;
                case BINDER_WORK_NODE: {
                        struct binder_node *node = container_of(w, struct binder_node, work);
                        uint32_t cmd = BR_NOOP;
                        const char *cmd_name;
                        int strong = node->internal_strong_refs || node->local_strong_refs;
                        int weak = !hlist_empty(&node->refs) || node->local_weak_refs || strong;
                        if (weak && !node->has_weak_ref) {
                                cmd = BR_INCREFS;
                                cmd_name = "BR_INCREFS";
                                node->has_weak_ref = 1;
                                node->pending_weak_ref = 1;
                                node->local_weak_refs++;
                        } else if (strong && !node->has_strong_ref) {
```

```
                cmd = BR_ACQUIRE;
                cmd_name = "BR_ACQUIRE";
                node->has_strong_ref = 1;
                node->pending_strong_ref = 1;
                node->local_strong_refs++;
            } else if (!strong && node->has_strong_ref) {
                cmd = BR_RELEASE;
                cmd_name = "BR_RELEASE";
                node->has_strong_ref = 0;
            } else if (!weak && node->has_weak_ref) {
                cmd = BR_DECREFS;
                cmd_name = "BR_DECREFS";
                node->has_weak_ref = 0;
            }
            if (cmd != BR_NOOP) {
                if (put_user(cmd, (uint32_t __user *)ptr))
                    return -EFAULT;
                ptr += sizeof(uint32_t);
                if (put_user(node->ptr, (void * __user *)ptr))
                    return -EFAULT;
                ptr += sizeof(void *);
                if (put_user(node->cookie, (void * __user *)ptr))
                    return -EFAULT;
                ptr += sizeof(void *);
                binder_stat_br(proc, thread, cmd);
                binder_debug(BINDER_DEBUG_USER_REFS,
                        "binder: %d:%d %s %d u%p c%p\n",
                        proc->pid, thread->pid, cmd_name, node->debug_id, node->
ptr, node->cookie);
            } else {
                list_del_init(&w->entry);
                if (!weak && !strong) {
                    binder_debug(BINDER_DEBUG_INTERNAL_REFS,
                            "binder: %d:%d node %d u%p c%p deleted\n",
                            proc->pid, thread->pid, node->debug_id,
                            node->ptr, node->cookie);
                    rb_erase(&node->rb_node, &proc->nodes);
                    kfree(node);
                    binder_stats_deleted(BINDER_STAT_NODE);
                } else {
                    binder_debug(BINDER_DEBUG_INTERNAL_REFS,
                            "binder: %d:%d node %d u%p c%p state unchanged\n",
                            proc->pid, thread->pid, node->debug_id, node->ptr,
                            node->cookie);
                }
            }
        } break;
        case BINDER_WORK_DEAD_BINDER:
        case BINDER_WORK_DEAD_BINDER_AND_CLEAR:
        case BINDER_WORK_CLEAR_DEATH_NOTIFICATION: {
            struct binder_ref_death *death;
```

```
                uint32_t cmd;
                death = container_of(w, struct binder_ref_death, work);
                if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION)
                        cmd = BR_CLEAR_DEATH_NOTIFICATION_DONE;
                else
                        cmd = BR_DEAD_BINDER;
                if (put_user(cmd, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
                if (put_user(death->cookie, (void * __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(void *);
                binder_stat_br(proc, thread, cmd);
                binder_debug(BINDER_DEBUG_DEATH_NOTIFICATION,
                             "binder: %d:%d %s %p\n",
                              proc->pid, thread->pid,
                              cmd == BR_DEAD_BINDER ?
                              "BR_DEAD_BINDER" :
                              "BR_CLEAR_DEATH_NOTIFICATION_DONE",
                              death->cookie);
                if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION) {
                        list_del(&w->entry);
                        kfree(death);
                        binder_stats_deleted(BINDER_STAT_DEATH);
                } else
                        list_move(&w->entry, &proc->delivered_death);
                if (cmd == BR_DEAD_BINDER)
                        goto done; /* DEAD_BINDER notifications can cause transactions */
        } break;
        }
        if (!t)
                continue;
        BUG_ON(t->buffer == NULL);
//把事务项 t 中的数据复制到本地局部变量 struct binder_transaction_data tr 中
        if (t->buffer->target_node) {
                struct binder_node *target_node = t->buffer->target_node;
                tr.target.ptr = target_node->ptr;
                tr.cookie =   target_node->cookie;
                t->saved_priority = task_nice(current);
                if (t->priority < target_node->min_priority &&
                    !(t->flags & TF_ONE_WAY))
                        binder_set_nice(t->priority);
                else if (!(t->flags & TF_ONE_WAY) ||
                         t->saved_priority > target_node->min_priority)
                        binder_set_nice(target_node->min_priority);
                cmd = BR_TRANSACTION;
        } else {
                tr.target.ptr = NULL;
                tr.cookie = NULL;
                cmd = BR_REPLY;
        }
```

```
                tr.code = t->code;
                tr.flags = t->flags;
                tr.sender_euid = t->sender_euid;
                if (t->from) {
                        struct task_struct *sender = t->from->proc->tsk;
                        tr.sender_pid = task_tgid_nr_ns(sender,
                                                        current->nsproxy->pid_ns);
                } else {
                        tr.sender_pid = 0;
                }
                tr.data_size = t->buffer->data_size;
                tr.offsets_size = t->buffer->offsets_size;
//t->buffer->data 所指向的地址是内核空间的，如果要把数据返回给 Service Manager 进程的用户空间
//而 Service Manager 进程的用户空间是不能访问内核空间数据的，所以需要进一步处理
//在具体处理时，Binder 机制使用类似浅复制的方法，通过在用户空间分配一个虚拟地址
//然后让这个用户空间虚拟地址与 t->buffer->data 这个内核空间虚拟地址指向同一个物理地址
//在此只需将 t->buffer->data 加上一个偏移值 proc->user_buffer_offset
//就可以得到 t->buffer->data 对应的用户空间虚拟地址了
//在调整了 tr.data.ptr.buffer 值后，需要一起调整 tr.data.ptr.offsets 的值
                tr.data.ptr.buffer = (void *)t->buffer->data +
                                        proc->user_buffer_offset;
                tr.data.ptr.offsets = tr.data.ptr.buffer +
                                        ALIGN(t->buffer->data_size,
                                           sizeof(void *));
//把 tr 的内容复制到用户传进来的缓冲区中，指针 ptr 指向这个用户缓冲区的地址
                if (put_user(cmd, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
                if (copy_to_user(ptr, &tr, sizeof(tr)))
                        return -EFAULT;
                ptr += sizeof(tr);
//上述代码只是对 tr.data.ptr.bufferr 和 tr.data.ptr.offsets 的内容进行了浅复制工作
                trace_binder_transaction_received(t);
                binder_stat_br(proc, thread, cmd);
                binder_debug(BINDER_DEBUG_TRANSACTION,
                             "binder: %d:%d %s %d %d:%d, cmd %d"
                             "size %zd-%zd ptr %p-%p\n",
                             proc->pid, thread->pid,
                             (cmd == BR_TRANSACTION) ? "BR_TRANSACTION" :
                             "BR_REPLY",
                             t->debug_id, t->from ? t->from->proc->pid : 0,
                             t->from ? t->from->pid : 0, cmd,
                             t->buffer->data_size, t->buffer->offsets_size,
                             tr.data.ptr.buffer, tr.data.ptr.offsets);
//从 todo 列表中删除，因为已经处理了这个事务
                list_del(&t->work.entry);
                t->buffer->allow_user_free = 1;
//如果 cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)为 true
//说明虽然在驱动程序中已经处理完了这个事务，但是仍然要在 Service Manager 完成之后需要等待回复确认
                if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
//把当前事务 t 放在 thread->transaction_stack 队列的头部
```

```
                    t->to_parent = thread->transaction_stack;
                    t->to_thread = thread;
                    thread->transaction_stack = t;
               }
//如果 cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)为 false
//则不需要等待回复了，而是直接删除事务 t
                    else {
                    t->buffer->transaction = NULL;
                    kfree(t);
                    binder_stats_deleted(BINDER_STAT_TRANSACTION);
               }
               break;
          }
done:
          *consumed = ptr - buffer;
          if (proc->requested_threads + proc->ready_threads == 0 &&
               proc->requested_threads_started < proc->max_threads &&
               (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
                BINDER_LOOPER_STATE_ENTERED)) /* the user-space code fails to */
                /*spawn a new thread if we leave this out */) {
                    proc->requested_threads++;
                    binder_debug(BINDER_DEBUG_THREADS,
                              "binder: %d:%d BR_SPAWN_LOOPER\n",
                              proc->pid, thread->pid);
                    if (put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffer))
                         return -EFAULT;
                    binder_stat_br(proc, thread, BR_SPAWN_LOOPER);
          }
          return 0;
}
```

由此可见，Binder 驱动程序是通过如下 4 个协议来引用运行在 Server 进程中的 Binder 本地对象的。

- ☑ BR_INCREFS。
- ☑ BR_ACQUIRE。
- ☑ BR_DECREFS。
- ☑ BR_RELEASE。

5.5.2　处理接口协议

在文件 frameworks/native/libs/binder/IPCThreadState.cpp 中，通过使用类成员函数 executeCommand()来处理 5.5.1 节中介绍的 4 个接口协议。函数 executeCommand()的具体实现代码如下所示。

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
     BBinder* obj;
     RefBase::weakref_type* refs;
     status_t result = NO_ERROR;

     switch (cmd) {
     case BR_ERROR:
          result = mIn.readInt32();
```

```
        break;

    case BR_OK:
        break;

    case BR_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
                    "BR_ACQUIRE: object %p does not match cookie %p (expected %p)",
                    refs, obj, refs->refBase());
        obj->incStrong(mProcess.get());
        IF_LOG_REMOTEREFS() {
            LOG_REMOTEREFS("BR_ACQUIRE from driver on %p", obj);
            obj->printRefs();
        }
        mOut.writeInt32(BC_ACQUIRE_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_RELEASE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
                    "BR_RELEASE: object %p does not match cookie %p (expected %p)",
                    refs, obj, refs->refBase());
        IF_LOG_REMOTEREFS() {
            LOG_REMOTEREFS("BR_RELEASE from driver on %p", obj);
            obj->printRefs();
        }
        mPendingStrongDerefs.push(obj);
        break;

    case BR_INCREFS:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        refs->incWeak(mProcess.get());
        mOut.writeInt32(BC_INCREFS_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_DECREFS:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        mPendingWeakDerefs.push(refs);
        break;

    case BR_ATTEMPT_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
```

```
        obj = (BBinder*)mIn.readInt32();

        {
            const bool success = refs->attemptIncStrong(mProcess.get());
            ALOG_ASSERT(success && refs->refBase() == obj,
                        "BR_ATTEMPT_ACQUIRE: object %p does not match cookie %p (expected %p)",
                        refs, obj, refs->refBase());

            mOut.writeInt32(BC_ACQUIRE_RESULT);
            mOut.writeInt32((int32_t)success);
        }
        break;

    case BR_TRANSACTION:
        {
            binder_transaction_data tr;
            result = mIn.read(&tr, sizeof(tr));
            ALOG_ASSERT(result == NO_ERROR,
                "Not enough command data for brTRANSACTION");
            if (result != NO_ERROR) break;

            Parcel buffer;
            buffer.ipcSetDataReference(
                reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                tr.data_size,
                reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                tr.offsets_size/sizeof(size_t), freeBuffer, this);

            const pid_t origPid = mCallingPid;
            const uid_t origUid = mCallingUid;

            mCallingPid = tr.sender_pid;
            mCallingUid = tr.sender_euid;

            int curPrio = getpriority(PRIO_PROCESS, mMyThreadId);
            if (gDisableBackgroundScheduling) {
                if (curPrio > ANDROID_PRIORITY_NORMAL) {
                    setpriority(PRIO_PROCESS, mMyThreadId, ANDROID_PRIORITY_NORMAL);
                }
            } else {
                if (curPrio >= ANDROID_PRIORITY_BACKGROUND) {
                    set_sched_policy(mMyThreadId, SP_BACKGROUND);
                }
            }

            //ALOGI(">>>> TRANSACT from pid %d uid %d\n", mCallingPid, mCallingUid);

            Parcel reply;
            IF_LOG_TRANSACTIONS() {
                TextOutput::Bundle _b(alog);
                alog << "BR_TRANSACTION thr " << (void*)pthread_self()
                    << " / obj " << tr.target.ptr << " / code "
```

```
                    << TypeCode(tr.code) << ": " << indent << buffer
                    << dedent << endl
                    << "Data addr = "
                    << reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer)
                    << ", offsets addr="
                    << reinterpret_cast<const size_t*>(tr.data.ptr.offsets) << endl;
            }
            if (tr.target.ptr) {
                sp<BBinder> b((BBinder*)tr.cookie);
                const status_t error = b->transact(tr.code, buffer, &reply, tr.flags);
                if (error < NO_ERROR) reply.setError(error);

            } else {
                const status_t error = the_context_object->transact(tr.code, buffer, &reply, tr.flags);
                if (error < NO_ERROR) reply.setError(error);
            }

            //ALOGI("<<<< TRANSACT from pid %d restore pid %d uid %d\n",
            //     mCallingPid, origPid, origUid);

            if ((tr.flags & TF_ONE_WAY) == 0) {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
                sendReply(reply, 0);
            } else {
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
            }

            mCallingPid = origPid;
            mCallingUid = origUid;

            IF_LOG_TRANSACTIONS() {
                TextOutput::Bundle _b(alog);
                alog << "BC_REPLY thr " << (void*)pthread_self() << " / obj "
                    << tr.target.ptr << ": " << indent << reply << dedent << endl;
            }

        }
        break;

    case BR_DEAD_BINDER:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
            proxy->sendObituary();
            mOut.writeInt32(BC_DEAD_BINDER_DONE);
            mOut.writeInt32((int32_t)proxy);
        } break;

    case BR_CLEAR_DEATH_NOTIFICATION_DONE:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
            proxy->getWeakRefs()->decWeak(proxy);
```

```
            } break;

        case BR_FINISHED:
            result = TIMED_OUT;
            break;

        case BR_NOOP:
            break;

        case BR_SPAWN_LOOPER:
            mProcess->spawnPooledThread(false);
            break;

        default:
            printf("*** BAD COMMAND %d received from Binder driver\n", cmd);
            result = UNKNOWN_ERROR;
            break;
        }
        if (result != NO_ERROR) {
            mLastError = result;
        }
        return result;
}
```

通过上述代码可知，在函数 executeCommand()中会调用 BBinder::transact 来处理 Client 端的请求。当需要多个线程提供服务时，驱动会请求创建新线程。具体创建某线程的过程，可以通过上述函数处理 BR_SPAWN_LOOPER 协议的过程获得。

5.6　引用对象 binder_ref 驱动

在 Android 系统中，引用对象的类型为 binder_ref 结构体，定义结构体 binder_ref 的代码如下所示。

```
struct binder_ref {
    //调试 ID
    int debug_id;
    //挂载到宿主对象 binder_proc 的红黑树 refs_by_desc 中的节点
    struct rb_node rb_node_desc;
    //挂载到宿主对象 binder_proc 的红黑树 refs_by_node 中的节点
    struct rb_node rb_node_node;
    //挂载到 Binder 实体对象的 refs 链表中的节点
    struct hlist_node node_entry;
    //Binder 引用对象的宿主进程 binder_proc
    struct binder_proc *proc;
    //Binder 引用对象所引用的 Binder 实体对象
    struct binder_node *node;
    //Binder 引用对象的句柄值
    uint32_t desc;
    //强引用计数
    int strong;
    //弱引用计数
```

```
        int weak;
        //注册死亡接收通知
        struct binder_ref_death *death;
};
```

在 Binder 机制中，binder_ref 用来描述一个 Binder 引用对象，每一个 client 在 Binder 驱动中都有一个 binder 引用对象。各个成员变量的具体说明如下。

☑ 成员变量 node：保存了该 Binder 引用对象所引用的 Binder 实体对象，Binder 实体对象使用链表保存了所有引用该实体对象的 Binder 引用对象。
☑ node_entry：是该 Binder 引用对象所引用的 Binder 实体对象的成员变量 refs 链表中的节点。
☑ desc：是一个句柄值，用来描述一个 Binder 引用对象。
☑ node：当 Client 进程通过句柄值来访问某个 Service 服务时，Binder 驱动程序可以通过该句柄值找到对应的 Binder 引用对象，然后根据该 Binder 引用对象的成员变量 node 找到对应的 Binder 实体对象，最后通过该 Binder 实体对象找到要访问的 Service。
☑ proc：执行该 Binder 引用对象的宿主进程。
☑ rb_node_desc 和 rb_node_node：是 binder_proc 中红黑树 refs_by_desc 和 refs_by_node 的节点。

Binder 驱动程序存在如下 4 个重要的协议，用于增加和减少 Binder 引用对象的强引用计数和弱引用计数。

☑ BR_INCREFS。
☑ BR_ACQUIRE。
☑ BR_DECREFS。
☑ BR_RELEASE。

上述计数处理功能通过函数 binder_thread_write()实现，此函数在文件/drivers/staging/android/binder.c 中定义，已经在本章前面的内容中进行了详细分析。

协议 BR_INCREFS 和 BR_ACQUIRE 用于增加一个 Binder 引用对象的强引用计数和弱引用计数，此功能是通过调用函数 binder_inc_ref()实现的，具体实现代码如下所示。

```
static int binder_inc_ref(struct binder_ref *ref, int strong,
                          struct list_head *target_list)
{
        int ret;
        if (strong) {
                if (ref->strong == 0) {
                        ret = binder_inc_node(ref->node, 1, 1, target_list);
                        if (ret)
                                return ret;
                }
                ref->strong++;
        } else {
                if (ref->weak == 0) {
                        ret = binder_inc_node(ref->node, 0, 1, target_list);
                        if (ret)
                                return ret;
                }
                ref->weak++;
        }
        return 0;
}
```

而协议 BR_RELEASE 和 BR_DECREFS 用于减少一个 Binder 引用对象的强引用计数和弱引用计数，此

功能是通过调用函数 binder_dec_ref 定义的，具体实现代码如下所示。

```
static int binder_dec_ref(struct binder_ref *ref, int strong)
{
	if (strong) {
		if (ref->strong == 0) {
			binder_user_error("binder: %d invalid dec strong, "
					  "ref %d desc %d s %d w %d\n",
					  ref->proc->pid, ref->debug_id,
					  ref->desc, ref->strong, ref->weak);
			return -EINVAL;
		}
		ref->strong--;
		if (ref->strong == 0) {
			int ret;
			ret = binder_dec_node(ref->node, strong, 1);
			if (ret)
				return ret;
		}
	} else {
		if (ref->weak == 0) {
			binder_user_error("binder: %d invalid dec weak, "
					  "ref %d desc %d s %d w %d\n",
					  ref->proc->pid, ref->debug_id,
					  ref->desc, ref->strong, ref->weak);
			return -EINVAL;
		}
		ref->weak--;
	}
	if (ref->strong == 0 && ref->weak == 0)
		binder_delete_ref(ref);
	return 0;
}
```

函数 binder_delete_ref 的功能是销毁 binder_ref 对象，具体实现代码如下所示。

```
static void binder_delete_ref(struct binder_ref *ref)
{
	binder_debug(BINDER_DEBUG_INTERNAL_REFS,
		     "binder: %d delete ref %d desc %d for "
		     "node %d\n", ref->proc->pid, ref->debug_id,
		     ref->desc, ref->node->debug_id);

	rb_erase(&ref->rb_node_desc, &ref->proc->refs_by_desc);
	rb_erase(&ref->rb_node_node, &ref->proc->refs_by_node);
	if (ref->strong)
		binder_dec_node(ref->node, 1, 1);
	hlist_del(&ref->node_entry);
	binder_dec_node(ref->node, 0, 1);
	if (ref->death) {
		binder_debug(BINDER_DEBUG_DEAD_BINDER,
			     "binder: %d delete ref %d desc %d "
			     "has death notification\n", ref->proc->pid,
			     ref->debug_id, ref->desc);
```

```
                list_del(&ref->death->work.entry);
                kfree(ref->death);
                binder_stats_deleted(BINDER_STAT_DEATH);
        }
        kfree(ref);
        binder_stats_deleted(BINDER_STAT_REF);
}
```

5.7 代理对象 BpBinder 驱动

在 Android 系统中，代理对象 BpBinder 是远程对象在当前进程的代理，它实现了 IBinder 接口。BBinder 与 BpBinder 很好区分，对于 Service 来说继承了 BBinder（BnInterface），因为 BBinder 有 onTransact 消息处理函数。而对于与 Service 通信的 Client 来说，需要继承 BpBinder(BpInterface)，因为 BpBinder 有消息传递函数 transcat。本节将详细讲解 Android 5.0 代理对象 BpBinder 驱动的核心知识。

5.7.1 创建 Binder 代理对象

以 cameraService 的 client 为例，文件 Camera.cpp 中的函数 getCameraService 能够获取远程 CameraService 的 IBinder 对象，然后通过如下代码进行重构，得到 BpCameraService 对象。

```
mCameraService = interface_cast<ICameraService>(binder);
```

而 BpCameraService 继承了 BpInterface，并传入了 BBinder。

```
cameraService：
    defaultServiceManager()->addService(
            String16("media.camera"), new CameraService());
```

在 IPC 传递的过程中，IBinder 指针不可缺少。这个指针对一个进程来说就像是 socket 的 ID 一样，是唯一的。无论这个 IBinder 是 BBinder 还是 BpBinder，它们都是在重构 BpBinder 或者 BBinder 时把 IBinder 作为参数传入。

在 Android 系统中，创建 Binder 代理对象在文件 frameworks\native\libs\binder\BpBinder.cpp 中定义，具体实现代码如下所示。

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
    ALOGV("Creating BpBinder %p handle %d\n", this, mHandle);
//设置 Binder 代理对象的生命周期收到弱引用的计数影响
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);
//调用当前线程内部的 IPCThreadState 的成员函数 incWeakHandle()增加相应的 Binder 引用对象的弱引用计数
    IPCThreadState::self()->incWeakHandle(handle);
}
```

在文件 frameworks/native/libs/binder/IPCThreadState.cpp 中，定义成员函数 incWeakHandle()的代码如下所示。

```
void IPCThreadState::incWeakHandle(int32_t handle)
{
```

```
    LOG_REMOTEREFS("IPCThreadState::incWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_INCREFS);
    mOut.writeInt32(handle);
}
```

5.7.2　销毁 Binder 代理对象

当销毁一个 Binder 代理对象时，线程会调用内部的 IPCThreadState 对象的成员函数 decWeakHandle()来减少相应的 Binder 引用对象的弱引用计数。函数 decWeakHandle()的具体实现代码如下所示。

```
void IPCThreadState::decWeakHandle(int32_t handle)
{
    LOG_REMOTEREFS("IPCThreadState::decWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_DECREFS);
    mOut.writeInt32(handle);
}
```

在 Binder 代理对象中，其 transact()函数的实现代码如下所示。

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }
    return DEAD_OBJECT;
}
```

各个参数的具体说明如下。

- ☑ code：表示请求的 ID 号。
- ☑ data：表示请求的参数。
- ☑ reply：表示返回的结果。
- ☑ flags：表示一些额外的标识，如 FLAG_ONEWAY，通常为 0。

上述函数 transact()只是简单地调用了 IPCThreadState::self()的 transact，在 IPCThreadState::transact 中的定义代码如下所示。

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err = data.errorCheck();

    flags |= TF_ACCEPT_FDS;

    IF_LOG_TRANSACTIONS() {
        TextOutput::Bundle _b(alog);
        alog << "BC_TRANSACTION thr " << (void*)pthread_self() << " / hand "
            << handle << " / code " << TypeCode(code) << ": "
            << indent << data << dedent << endl;
```

```
    }

    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }

        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_REPLY thr " << (void*)pthread_self() << " / hand "
                << handle << ": ";
            if (reply) alog << indent << *reply << dedent << endl;
            else alog << "(none requested)" << endl;
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }

    return err;
}

status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    int32_t cmd;
    int32_t err;

    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;

        cmd = mIn.readInt32();

        IF_LOG_COMMANDS() {
            alog << "Processing waitForResponse Command: "
                << getReturnString(cmd) << endl;
```

```
    }

    switch (cmd) {
    case BR_TRANSACTION_COMPLETE:
        if (!reply && !acquireResult) goto finish;
        break;

    case BR_DEAD_REPLY:
        err = DEAD_OBJECT;
        goto finish;

    case BR_FAILED_REPLY:
        err = FAILED_TRANSACTION;
        goto finish;

    case BR_ACQUIRE_RESULT:
        {
            LOG_ASSERT(acquireResult != NULL, "Unexpected brACQUIRE_RESULT");
            const int32_t result = mIn.readInt32();
            if (!acquireResult) continue;
            *acquireResult = result ? NO_ERROR : INVALID_OPERATION;
        }
        goto finish;

    case BR_REPLY:
        {
            binder_transaction_data tr;
            err = mIn.read(&tr, sizeof(tr));
            LOG_ASSERT(err == NO_ERROR, "Not enough command data for brREPLY");
            if (err != NO_ERROR) goto finish;

            if (reply) {
                if ((tr.flags & TF_STATUS_CODE) == 0) {
                    reply->ipcSetDataReference(
                        reinterpret_cast(tr.data.ptr.buffer),
                        tr.data_size,
                        reinterpret_cast(tr.data.ptr.offsets),
                        tr.offsets_size/sizeof(size_t),
                        freeBuffer, this);
                } else {
                    err = *static_cast(tr.data.ptr.buffer);
                    freeBuffer(NULL,
                        reinterpret_cast(tr.data.ptr.buffer),
                        tr.data_size,
                        reinterpret_cast(tr.data.ptr.offsets),
                        tr.offsets_size/sizeof(size_t), this);
                }
            } else {
                freeBuffer(NULL,
                    reinterpret_cast(tr.data.ptr.buffer),
                    tr.data_size,
```

```
                        reinterpret_cast(tr.data.ptr.offsets),
                        tr.offsets_size/sizeof(size_t), this);
                    continue;
                }
            }
            goto finish;

        default:
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
    }

finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
    }

    return err;
}
```

在上述代码中，通过内核模块 transact 将请求发送了给服务端。当服务端处理完请求之后，会沿着原路返回结果给调用者。在此可以看出请求是同步操作，它会等待直到结果返回为止。这样在 BpBinder 之上进行简单包装之后，就可以得到与服务对象相同的接口，调用者不需要关心调用的对象是远程的还是本地的。

第 6 章　Logger 驱动架构详解

Logger 驱动是 Android 系统中一个轻量级日志驱动，此驱动为用户层程序提供了日志记录的支持。在 Android 开发应用中，通常将 Logger 驱动作为一个工具来使用。本章将详细分析 Android 系统中 Logger 日志驱动的基本架构知识，为读者学习本书后面的知识打下基础。

6.1　分析 Logger 驱动程序

在 Android 系统中，Logger 系统有如下 3 个设备节点。

☑　/dev/log/main：主要的 log。

☑　/dev/log/events：事件的 log。

☑　/dev/log/radio：Modem 部分的 log。

Logger 驱动为用户空间提供了 ioctl 接口、read 接口和异步 write 接口，其主设备号为 10（Misc Driver），实现源代码位于 kernel/common/drivers/staging/android/logger.h 和 kernel/common/drivers/staging/android/logger.c 源文件中。

对于非本用户或本组而言，Logger 驱动程序的设备节点是可写而不可读的。在 Android 用户空间中，使用 liblog 库封装了 Logger 驱动程序，其路径为 system/core/liblog。logcat 程序负责调用 Logger 驱动，此程序是一个可知性程序，当用户取出系统 log 信息后在系统中使用的一个辅助工具，logcat 程序的代码路径为 system/core/logcat。

在 Android 内核源码中，Logger 驱动程序在 drivers/staging/android/logger.h 和 drivers/staging/android/logger.c 文件中实现。

本节将详细介绍 Android 系统中 Logger 驱动程序的基本知识。

6.1.1　分析头文件

头文件 logger.h 的具体实现代码如下所示。

```
#ifndef _LINUX_LOGGER_H
#define _LINUX_LOGGER_H
#include <linux/types.h>
#include <linux/ioctl.h>
struct user_logger_entry_compat {
		__u16		len;
		__u16		__pad;
		__s32		pid;
		__s32		tid;
		__s32		sec;
		__s32		nsec;
		char		msg[0];
```

```
};
struct logger_entry {
        __u16           len;
        __u16           hdr_size;
        __s32           pid;
        __s32           tid;
        __s32           sec;
        __s32           nsec;
        kuid_t          euid;
        char            msg[0];
};
#define LOGGER_LOG_RADIO            "log_radio"         /* radio-related messages */
#define LOGGER_LOG_EVENTS           "log_events"        /* system/hardware events */
#define LOGGER_LOG_SYSTEM           "log_system"        /* system/framework messages */
#define LOGGER_LOG_MAIN             "log_main"          /* everything else */
#define LOGGER_ENTRY_MAX_PAYLOAD            4076
0#define __LOGGERIO          0xAE
#define LOGGER_GET_LOG_BUF_SIZE          _IO(__LOGGERIO, 1) /* size of log */
#define LOGGER_GET_LOG_LEN               _IO(__LOGGERIO, 2) /* used log len */
#define LOGGER_GET_NEXT_ENTRY_LEN        _IO(__LOGGERIO, 3) /* next entry len */
#define LOGGER_FLUSH_LOG                 _IO(__LOGGERIO, 4) /* flush log */
#define LOGGER_GET_VERSION               _IO(__LOGGERIO, 5) /* abi version */
#define LOGGER_SET_VERSION               _IO(__LOGGERIO, 6) /* abi version */
#endif /* _LINUX_LOGGER_H */
```

上述代码的核心是结构体 struct logger_entry，功能是描述某条 Log 记录，各个成员变量的具体说明如下。

☑ 成员变量 len：记录了这条记录的有效负载的长度，有效负载指定的只是日志记录本身的长度，并不包括用于描述这个记录的 struct logger_entry 结构体。

☑ 成员变量 pid 和 tid：分别用来记录是哪条进程被写入到这条记录中。

☑ 成员变量 sec 和 nsec：用于记录日志写的时间。

☑ 成员变量 msg：用于记录有效负载的内容，其大小由 len 成员变量来确定。

另外在上述代码中还定义了两个宏，其中通过 LOGGER_ENTRY_MAX_PAYLOAD 设置每条日志记录的有效负载长度加上结构体 logger_entry 的长度不能超过 4076 个字节。

6.1.2 驱动实现文件

文件 logger.c 是头文件 logger.h 的具体实现，接下来将详细剖析其具体实现过程。

（1）看结构体 logger_log，具体实现代码如下所示。

```
struct logger_log {
        unsigned char           *buffer;
        struct miscdevice       misc;
        wait_queue_head_t       wq;
        struct list_head        readers;
        struct mutex            mutex;
        size_t                  w_off;
        size_t                  head;
        size_t                  size;
        struct list_head        logs;
};
```

结构体 logger_log 的功能是保存日志数据，这是真正保存的地方，各个成员变量的具体说明如下。

☑ 成员变量 buffer：用于保存日志信息的内存缓冲区，其大小由成员变量 size 确定。
☑ 成员变量 misc：描述了 logger 驱动程序使用的设备属于 misc 类型的设备，通过在 Android 模拟器上执行 cat/proc/devices 命令可以看出，misc 类型设备的主设备号是 10。
☑ 成员变量 wq：这是一个等待队列，用于保存正在等待读取日志的进程。
☑ 成员变量 readers：用于保存当前正在读取日志的进程，由结构体 logger_reader 来描述这些正在被读取的日志进程。
☑ 成员变量 mutex：这是一个互斥变量，用于保护 log 的并发访问。因为整个日志系的读写过程会产生“生产者-消费者”的问题，所以需要一个互斥量来保护 log 的并发访问。
☑ 成员变量 w_off：用于记录下一条日志应该从哪里开始写。
☑ 成员变量 head：设置在打开日志文件中从什么位置开始读取日志。

（2）再看结构体 logger_reader，具体实现代码如下所示。

```
struct logger_reader {
        struct logger_log       *log;
        struct list_head        list;
        size_t                  r_off;
        bool                    r_all;
        int                     r_ver;
};
```

结构体 logger_reader 的功能是表示一个读取日志的进程，各个成员变量的具体说明如下。

☑ 成员变量 log：用于指向要读取的日志缓冲区。
☑ 成员变量 list：用于连接其他读者进程。
☑ 成员变量 r_off：表示当前要读取的日志在缓冲区中的位置。

（3）在文件 logger.h 中实现了日志驱动的初始化操作，相关代码如下所示。

```
#define LOGGER_LOG_RADIO            "log_radio"
#define LOGGER_LOG_EVENTS           "log_events"
#define LOGGER_LOG_SYSTEM           "log_system"
#define LOGGER_LOG_MAIN             "log_main"

#define LOGGER_ENTRY_MAX_PAYLOAD         4076

#define __LOGGERIO        0xAE

#define LOGGER_GET_LOG_BUF_SIZE          _IO(__LOGGERIO, 1)
#define LOGGER_GET_LOG_LEN               _IO(__LOGGERIO, 2)
#define LOGGER_GET_NEXT_ENTRY_LEN        _IO(__LOGGERIO, 3)
#define LOGGER_FLUSH_LOG                 _IO(__LOGGERIO, 4)
#define LOGGER_GET_VERSION               _IO(__LOGGERIO, 5)
#define LOGGER_SET_VERSION               _IO(__LOGGERIO, 6)
```

在上述代码中定义了 3 个日志设备：log_main、log_events 和 log_radio，对应的名称分别 LOGGER_ LOG_ MAIN、LOGGER_LOG_EVENTS 和 LOGGER_LOG_RADIO，对应的次设备号为 MISC_DYNAMIC_ MINOR，用于在注册时动态分配。通过上述宏定义代码可以看出这 3 个日志设备的用途。

接下来看注册的日志设备文件的操作方法 logger_fops，具体实现代码如下。

```
static const struct file_operations logger_fops = {
        .owner = THIS_MODULE,
        .read = logger_read,
```

```
        .aio_write = logger_aio_write,
        .poll = logger_poll,
        .unlocked_ioctl = logger_ioctl,
        .compat_ioctl = logger_ioctl,
        .open = logger_open,
        .release = logger_release,
};
```

在上述代码中，指定了多个设备文件的操作方法，例如 logger_open、logger_read 和 logger_aio_write。

（4）开始进行具体的初始化，其中初始化函数为 logger_init()，具体实现代码如下所示。

```
static int __init logger_init(void)
{
        int ret;

        ret = create_log(LOGGER_LOG_MAIN, 256*1024);
        if (unlikely(ret))
                goto out;

        ret = create_log(LOGGER_LOG_EVENTS, 256*1024);
        if (unlikely(ret))
                goto out;

        ret = create_log(LOGGER_LOG_RADIO, 256*1024);
        if (unlikely(ret))
                goto out;

        ret = create_log(LOGGER_LOG_SYSTEM, 256*1024);
        if (unlikely(ret))
                goto out;

out:
        return ret;
}
```

在上述代码中调用了函数 create_log()，功能是依次初始化前面介绍的日志设备 LOGGER_LOG_MAIN、LOGGER_LOG_EVENTS、LOGGER_LOG_RADIO 和 LOGGER_LOG_SYSTEM。函数 create_log()的具体实现代码如下所示。

```
static int __init create_log(char *log_name, int size)
{
        int ret = 0;
        struct logger_log *log;
        unsigned char *buffer;

        buffer = vmalloc(size);
        if (buffer == NULL)
                return -ENOMEM;

        log = kzalloc(sizeof(struct logger_log), GFP_KERNEL);
        if (log == NULL) {
                ret = -ENOMEM;
                goto out_free_buffer;
        }
```

```
        log->buffer = buffer;

        log->misc.minor = MISC_DYNAMIC_MINOR;
        log->misc.name = kstrdup(log_name, GFP_KERNEL);
        if (log->misc.name == NULL) {
                ret = -ENOMEM;
                goto out_free_log;
        }

        log->misc.fops = &logger_fops;
        log->misc.parent = NULL;

        init_waitqueue_head(&log->wq);
        INIT_LIST_HEAD(&log->readers);
        mutex_init(&log->mutex);
        log->w_off = 0;
        log->head = 0;
        log->size = size;

        INIT_LIST_HEAD(&log->logs);
        list_add_tail(&log->logs, &log_list);

        /* finally, initialize the misc device for this log */
        ret = misc_register(&log->misc);
        if (unlikely(ret)) {
                pr_err("failed to register misc device for log '%s'!\n",
                                log->misc.name);
                goto out_free_log;
        }

        pr_info("created %luK log '%s'\n",
                (unsigned long) log->size >> 10, log->misc.name);

        return 0;

out_free_log:
        kfree(log);

out_free_buffer:
        vfree(buffer);
        return ret;
}
```

在上述代码中，通过调用函数 misc_register()注册了 misc 设备。

（5）注册 misc 设备。

函数 misc_register()在文件 kernel/common/drivers/char/misc.c 中定义，具体实现代码如下所示。

```
int misc_register(struct miscdevice * misc)
{
        dev_t dev;
        int err = 0;

```

```
	INIT_LIST_HEAD(&misc->list);

	mutex_lock(&misc_mtx);

	if (misc->minor == MISC_DYNAMIC_MINOR) {
		int i = find_first_zero_bit(misc_minors, DYNAMIC_MINORS);
		if (i >= DYNAMIC_MINORS) {
			mutex_unlock(&misc_mtx);
			return -EBUSY;
		}
		misc->minor = DYNAMIC_MINORS - i - 1;
		set_bit(i, misc_minors);
	} else {
		struct miscdevice *c;

		list_for_each_entry(c, &misc_list, list) {
			if (c->minor == misc->minor) {
				mutex_unlock(&misc_mtx);
				return -EBUSY;
			}
		}
	}

	dev = MKDEV(MISC_MAJOR, misc->minor);

	misc->this_device = device_create(misc_class, misc->parent, dev,
					  misc, "%s", misc->name);
	if (IS_ERR(misc->this_device)) {
		int i = DYNAMIC_MINORS - misc->minor - 1;
		if (i < DYNAMIC_MINORS && i >= 0)
			clear_bit(i, misc_minors);
		err = PTR_ERR(misc->this_device);
		goto out;
	}

	/*
	 * Add it to the front, so that later devices can "override"
	 * earlier defaults
	 */
	list_add(&misc->list, &misc_list);
out:
	mutex_unlock(&misc_mtx);
	return err;
}
```

在上述代码中，misc_register 在加载模块时会自动创建设备文件为主设备号为 10 的字符设备。在 Linux 内核的 include/linux/miscdevice.h 文件，要把自己定义的 misc device 从设备定义在这里。其实是因为这些字符设备不符合预先确定的字符设备范畴，所有这些设备采用主设备号 10，一起归于 misc device，其实 misc_register 就是用主设备号 10 调用 register_chrdev()的。也就是说，misc_device 是特殊的字符设备。注册驱动程序时采用 misc_register()函数注册，此函数中会自动创建设备节点，即设备文件，无须 mknod 指令创

建设备文件。因为 misc_register()会调用 class_device_create()或者 device_create()。

通过上述代码完成注册工作后，使用函数 device_create()创建设备文件节点。这里将创建/dev/log/main、/dev/log/events 和/dev/log/radio 3 个设备文件，这样，用户空间就可以通过读写这 3 个文件和驱动程序进行交互。

（6）打开日志设备，在读取日志驱动程序或写入日志记录之前，需要先调用函数 logger_open()打开对应的日志设备文件。函数 logger_open()的具体实现代码如下所示。

```
static int logger_open(struct inode *inode, struct file *file)
{
	struct logger_log *log;
	int ret;

	ret = nonseekable_open(inode, file);
	if (ret)
		return ret;

	log = get_log_from_minor(MINOR(inode->i_rdev));
	if (!log)
		return -ENODEV;

	if (file->f_mode & FMODE_READ) {
		struct logger_reader *reader;

		reader = kmalloc(sizeof(struct logger_reader), GFP_KERNEL);
		if (!reader)
			return -ENOMEM;

		reader->log = log;
		reader->r_ver = 1;
		reader->r_all = in_egroup_p(inode->i_gid) ||
			capable(CAP_SYSLOG);

		INIT_LIST_HEAD(&reader->list);

		mutex_lock(&log->mutex);
		reader->r_off = log->head;
		list_add_tail(&reader->list, &log->readers);
		mutex_unlock(&log->mutex);

		file->private_data = reader;
	} else
		file->private_data = log;

	return 0;
}
```

通过上述代码可知，在打开新的日志设备文件时，从 log->head 位置开始读取日志，并将结果保存在 struct logger_reader 的成员变量 r_off 中。使用 start 标注的 while 循环用于等待日志的可读性，如果已经没有新的可读日志，则读进程将要进入休眠状态，直到写入新的日志后再唤醒。上述功能是通过调用函数 prepare_wait()和 schedule()实现的。如果没有新的日志可读，并且设备文件不是以非阻塞 O_NONBLOCK 的方式打开或者这时有信号要处理（signal_pending(current)），那么就直接返回，不再等待新的日志写入。判断当前是否有

新的日志可读的代码片段是：

```
ret = (log->w_off == reader->r_off);
```

通过上述代码即可判断当前缓冲区的写入位置和当前读进程的读取位置是否相等，如果不等则说明有新的日志可读。

（7）在函数 logger_open()中调用了 get_log_from_minor()函数，功能是根据设备号获取要操作的日志的日志缓冲区结构体，具体实现代码如下所示。

```
static struct logger_log *get_log_from_minor(int minor)
{
	struct logger_log *log;

	list_for_each_entry(log, &log_list, logs)
		if (log->misc.minor == minor)
			return log;
	return NULL;
}
```

（8）通过函数 logger_read()读取日志设备中的记录数据，具体实现代码如下所示。

```
static ssize_t logger_read(struct file *file, char __user *buf,
				size_t count, loff_t *pos)
{
//将得到的日志记录读取进程结构体 reader
	struct logger_reader *reader = file->private_data;
//得到日志缓冲区的 log
	struct logger_log *log = reader->log;
	ssize_t ret;
	DEFINE_WAIT(wait);

start:
//使用循环检查日志缓冲区结构体 log 中是否有可读取的日志记录
	while (1) {
		mutex_lock(&log->mutex);

		prepare_to_wait(&log->wq, &wait, TASK_INTERRUPTIBLE);

		ret = (log->w_off == reader->r_off);
		mutex_unlock(&log->mutex);
		if (!ret)
			break;

		if (file->f_flags & O_NONBLOCK) {
			ret = -EAGAIN;
			break;
		}

		if (signal_pending(current)) {
			ret = -EINTR;
			break;
		}

		schedule();
```

```
	}

	finish_wait(&log->wq, &wait);
	if (ret)
		return ret;

	mutex_lock(&log->mutex);

	if (!reader->r_all)
		reader->r_off = get_next_entry_by_uid(log,
			reader->r_off, current_euid());

	/* is there still something to read or did we race? */
	if (unlikely(log->w_off == reader->r_off)) {
		mutex_unlock(&log->mutex);
		goto start;
	}

	/* get the size of the next entry */
	ret = get_user_hdr_len(reader->r_ver) +
		get_entry_msg_len(log, reader->r_off);
	if (count < ret) {
		ret = -EINVAL;
		goto out;
	}

	/* get exactly one entry from the log */
	ret = do_read_log_to_user(log, reader, buf, ret);

out:
	mutex_unlock(&log->mutex);

	return ret;
}
```

在上述代码中，如果在日志缓冲区结构体 log 中有可读取的日志记录，则调用函数 get_entry_msg_len()和 get_user_hdr_len()来获取下一条可读的日志记录的长度，这两个函数 get_entry_msg_len()的具体实现代码如下所示。

```
static __u32 get_entry_msg_len(struct logger_log *log, size_t off)
{
	struct logger_entry scratch;
	struct logger_entry *entry;

	entry = get_entry_header(log, off, &scratch);
	return entry->len;
}

static size_t get_user_hdr_len(int ver)
{
	if (ver < 2)
		return sizeof(struct user_logger_entry_compat);
```

```
	else
		return sizeof(struct logger_entry);
}
```

日志读取进程是以日志记录为单位进行读取的，一次只读取一条记录。日志系统中的每一条日志记录由两大部分组成，一个用于描述这条日志记录的结构体 logger_entry，另一个是记录体本身（即有效负载）。结构体 logger_entry 的长度是固定的，只要知道其有效的负载长度，就可以知道整条日志记录的长度。而有效负载的长度是在结构体 logger_entry 的成员变量 len 中记录，成员变量 len 的地址与结构体 logger_entry 的地址相同，所以只需读取记录的开始位置的两个字节即可。

因为日志记录缓冲区是循环使用的，所以会发生如下两种情形。

☑ 第一种：这两个字节有可能是第一个字节存放在缓冲区最后一个字节，第二个字节存放在缓冲区的第一个字节。此时分别通过读取缓冲区最后一个字节和第一个字节来得到日志记录的有效负载长度到本地变量 val 中。

☑ 第二种：这两个字节都是连在一起的。此时直接读取连续两个字节的值到本地变量 val 中。

对于上述两种情形来说，是通过判断日志缓冲区的大小和要读取的日志记录在缓冲区中的位置的差值来区别的，如果相差 1 则说明是第一种情况。只要把有效负载的长度 val 加上结构体 logger_entry 的长度后，就会得到要读取的日志记录的总长度。

（9）执行真正的读取操作。在前面介绍的函数 logger_read()中，调用了函数 do_read_log_to_user()来执行真正的读取操作，此函数的具体实现代码如下所示。

```
static ssize_t do_read_log_to_user(struct logger_log *log,
				   struct logger_reader *reader,
				   char __user *buf,
				   size_t count)
{
	struct logger_entry scratch;
	struct logger_entry *entry;
	size_t len;
	size_t msg_start;

	/*
	 * First, copy the header to userspace, using the version of
	 * the header requested
	 */
	entry = get_entry_header(log, reader->r_off, &scratch);
	if (copy_header_to_user(reader->r_ver, entry, buf))
		return -EFAULT;

	count -= get_user_hdr_len(reader->r_ver);
	buf += get_user_hdr_len(reader->r_ver);
	msg_start = logger_offset(log,
		reader->r_off + sizeof(struct logger_entry));

	/*
	 * We read from the msg in two disjoint operations. First, we read from
	 * the current msg head offset up to 'count' bytes or to the end of
	 * the log, whichever comes first.
	 */
	len = min(count, log->size - msg_start);
```

```
		if (copy_to_user(buf, log->buffer + msg_start, len))
			return -EFAULT;

		/*
		 * Second, we read any remaining bytes, starting back at the head of
		 * the log.
		 */
		if (count != len)
			if (copy_to_user(buf + len, log->buffer, count - len))
				return -EFAULT;

		reader->r_off = logger_offset(log, reader->r_off +
			sizeof(struct logger_entry) + count);

		return count + get_user_hdr_len(reader->r_ver);
}
```

在上述代码中，通过函数 copy_to_user()将内核空间的日志缓冲区指定的内容复制到了用户空间的内存缓冲区，并把当前读取日志进程的上下文信息中的读偏移 r_off 前进到下一条日志记录的开始位置。

（10）再看日志记录数据的过程，此过程通过调用函数 logger_aio_write()实现，具体实现代码如下所示。

```
static ssize_t logger_aio_write(struct kiocb *iocb, const struct iovec *iov,
				unsigned long nr_segs, loff_t ppos)
{
		struct logger_log *log = file_get_log(iocb->ki_filp);
		size_t orig;
		struct logger_entry header;
		struct timespec now;
		ssize_t ret = 0;

		now = current_kernel_time();

		header.pid = current->tgid;
		header.tid = current->pid;
		header.sec = now.tv_sec;
		header.nsec = now.tv_nsec;
		header.euid = current_euid();
		header.len = min_t(size_t, iocb->ki_nbytes, LOGGER_ENTRY_MAX_PAYLOAD);
		header.hdr_size = sizeof(struct logger_entry);

		/* null writes succeed, return zero */
		if (unlikely(!header.len))
			return 0;

		mutex_lock(&log->mutex);

		orig = log->w_off;

		/*
		 * Fix up any readers, pulling them forward to the first readable
		 * entry after (what will be) the new write offset. We do this now
		 * because if we partially fail, we can end up with clobbered log
```

```
         * entries that encroach on readable buffer
         */
        fix_up_readers(log, sizeof(struct logger_entry) + header.len);

        do_write_log(log, &header, sizeof(struct logger_entry));

        while (nr_segs-- > 0) {
                size_t len;
                ssize_t nr;

                /* figure out how much of this vector we can keep */
                len = min_t(size_t, iov->iov_len, header.len - ret);

                /* write out this segment's payload */
                nr = do_write_log_from_user(log, iov->iov_base, len);
                if (unlikely(nr < 0)) {
                        log->w_off = orig;
                        mutex_unlock(&log->mutex);
                        return nr;
                }

                iov++;
                ret += nr;
        }

        mutex_unlock(&log->mutex);

        /* wake up any blocked readers */
        wake_up_interruptible(&log->wq);

        return ret;
}
```

在上述代码中，参数 iocb 表示 I/O 的上下文，参数 iov 表示要写入的内容，长度为 nr_segs，表示要写入 nr_segs 个段的内容。每个要写入的日志的结构格式为：

```
struct logger_entry | priority | tag | msg
```

在上述格式中，priority、tag 和 msg 的内容是由参数 iov 从用户空间传递下来的，分别对应 iov 中的 3 个元素，而 logger_entry 是由内核空间构造实现的。

（11）获取对应日志缓冲区结构体。在函数 logger_aio_write()中还调用了函数 file_get_log()安全地获取了对应的日志缓冲区结构体。函数 file_get_log 的具体实现代码如下所示。

```
static inline struct logger_log *file_get_log(struct file *file)
{
        if (file->f_mode & FMODE_READ) {
                struct logger_reader *reader = file->private_data;
                return reader->log;
        } else
                return file->private_data;
}
```

Android 系统的日志缓冲区是循环使用的，也就是说如果没有及时读取旧的日志记录，而缓冲区的内容又已经被用完时，就需要覆盖旧的记录以容纳新的记录。而这部分将要被覆盖的内容可能是某些 reader 的

下一次要读取的日志所在的位置，以及为新的 reader 准备的日志开始读取位置 head 所在的位置。基于上述原因，需要做一些调整位置的工作，目的是使它们能够指向一个新的有效的位置。为此特意编写了函数 fix_up_reader()以实现上述功能，此函数的具体实现代码如下所示。

```
static void fix_up_readers(struct logger_log *log, size_t len)
{
	size_t old = log->w_off;
	size_t new = logger_offset(log, old + len);
	struct logger_reader *reader;

	if (is_between(old, new, log->head))
		log->head = get_next_entry(log, log->head, len);

	list_for_each_entry(reader, &log->readers, list)
		if (is_between(old, new, reader->r_off))
			reader->r_off = get_next_entry(log, reader->r_off, len);
}
```

在上述代码中，参数 len 表示将要写入的日志记录的长度。判断 log->head 和所有读者 reader 的当前读偏移 reader->r_off 是否在被覆盖的区域内，如果是则调用函数 get_next_entry 来获取下一个有效记录的起始位置以调整当前位置。函数 get_next_entry()的具体实现代码如下所示。

```
static size_t get_next_entry(struct logger_log *log, size_t off, size_t len)
{
	size_t count = 0;

	do {
		size_t nr = sizeof(struct logger_entry) +
			get_entry_msg_len(log, off);
		off = logger_offset(log, off + nr);
		count += nr;
	} while (count < len);

	return off;
}
```

在函数 fix_up_reader()中需要判断 c 是否位于 a 和 b 之间，此功能通过函数 is_between()实现，具体实现代码如下所示。

```
static inline int is_between(size_t a, size_t b, size_t c)
{
//如果 a 小于 b
	if (a < b) {
		/* is c between a and b? */
		if (a < c && c <= b)
			return 1;
	} else {
		/* is c outside of b through a? */
		if (c <= b || a < c)
			return 1;
	}

	return 0;
}
```

当为将要写入的日志记录准备好日志记录结构体 header 后，下一步即可调用函数 do_write_log()将其内容写入日志缓冲区结构体 log 中，函数 do_write_log 的具体实现代码如下所示。

```
static void do_write_log(struct logger_log *log, const void *buf, size_t count)
{
	size_t len;

	len = min(count, log->size - log->w_off);
	memcpy(log->buffer + log->w_off, buf, len);

	if (count != len)
		memcpy(log->buffer, buf + len, count - len);

	log->w_off = logger_offset(log, log->w_off + count);

}
```

（12）复制参数保存的内容。在整个写入过程中，需要调用函数 do_write_log_from_user()将参数（函数 logger_aio_write()的参数）iov 保存的内容复制到日志缓冲区结构体 log 中。函数 do_write_log_from_user()的具体实现代码如下所示。

```
static ssize_t do_write_log_from_user(struct logger_log *log,
				      const void __user *buf, size_t count)
{
	size_t len;

	len = min(count, log->size - log->w_off);
	if (len && copy_from_user(log->buffer + log->w_off, buf, len))
		return -EFAULT;

	if (count != len)
		if (copy_from_user(log->buffer, buf + len, count - len))
			/*
			 * Note that by not updating w_off, this abandons the
			 * portion of the new entry that *was* successfully
			 * copied, just above.   This is intentional to avoid
			 * message corruption from missing fragments.
			 */
			return -EFAULT;

	log->w_off = logger_offset(log, log->w_off + count);

	return count;
}
```

（13）函数 logger_poll()的功能是判断当前进程是否可以操作日志设备，具体实现代码如下所示。

```
static unsigned int logger_poll(struct file *file, poll_table *wait)
{
	struct logger_reader *reader;
	struct logger_log *log;
	unsigned int ret = POLLOUT | POLLWRNORM;

	if (!(file->f_mode & FMODE_READ))
```

```
                return ret;

        reader = file->private_data;
        log = reader->log;

        poll_wait(file, &log->wq, wait);

        mutex_lock(&log->mutex);
        if (!reader->r_all)
                reader->r_off = get_next_entry_by_uid(log,
                        reader->r_off, current_euid());

        if (log->w_off != reader->r_off)
                ret |= POLLIN | POLLRDNORM;
        mutex_unlock(&log->mutex);

        return ret;
}
```

通过上述代码可以看出，POLLOUT 总是成立的，即进程总是可以进行写入操作。读操作则不一样，如果只是以 FMODE_READ 模式打开日志设备的进程，那么就需要判断当前日志缓冲区是否为空，只有不为空才能读取日志。

再看函数 logger_ioctl()，具体实现代码如下所示。

```
static long logger_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        struct logger_log *log = file_get_log(file);
        struct logger_reader *reader;
        long ret = -EINVAL;
        void __user *argp = (void __user *) arg;

        mutex_lock(&log->mutex);

        switch (cmd) {
        case LOGGER_GET_LOG_BUF_SIZE:
                ret = log->size;
                break;
        case LOGGER_GET_LOG_LEN:
                if (!(file->f_mode & FMODE_READ)) {
                        ret = -EBADF;
                        break;
                }
                reader = file->private_data;
                if (log->w_off >= reader->r_off)
                        ret = log->w_off - reader->r_off;
                else
                        ret = (log->size - reader->r_off) + log->w_off;
                break;
        case LOGGER_GET_NEXT_ENTRY_LEN:
                if (!(file->f_mode & FMODE_READ)) {
                        ret = -EBADF;
                        break;
```

```
		}
		reader = file->private_data;

		if (!reader->r_all)
			reader->r_off = get_next_entry_by_uid(log,
				reader->r_off, current_euid());

		if (log->w_off != reader->r_off)
			ret = get_user_hdr_len(reader->r_ver) +
				get_entry_msg_len(log, reader->r_off);
		else
			ret = 0;
		break;
	case LOGGER_FLUSH_LOG:
		if (!(file->f_mode & FMODE_WRITE)) {
			ret = -EBADF;
			break;
		}
		if (!(in_egroup_p(file_inode(file)->i_gid) ||
				capable(CAP_SYSLOG))) {
			ret = -EPERM;
			break;
		}
		list_for_each_entry(reader, &log->readers, list)
			reader->r_off = log->w_off;
		log->head = log->w_off;
		ret = 0;
		break;
	case LOGGER_GET_VERSION:
		if (!(file->f_mode & FMODE_READ)) {
			ret = -EBADF;
			break;
		}
		reader = file->private_data;
		ret = reader->r_ver;
		break;
	case LOGGER_SET_VERSION:
		if (!(file->f_mode & FMODE_READ)) {
			ret = -EBADF;
			break;
		}
		reader = file->private_data;
		ret = logger_set_version(reader, argp);
		break;
	}

	mutex_unlock(&log->mutex);

	return ret;
}
```

```
static const struct file_operations logger_fops = {
        .owner = THIS_MODULE,
        .read = logger_read,
        .aio_write = logger_aio_write,
        .poll = logger_poll,
        .unlocked_ioctl = logger_ioctl,
        .compat_ioctl = logger_ioctl,
        .open = logger_open,
        .release = logger_release,
};
```

通过上述实现代码可知，函数 logger_ioctl()对一些命令进行了操作，可以支持如下命令操作。

☑ LOGGER_GET_LOG_BUF_SIZE：用于得到日志环形缓冲区的尺寸。
☑ LOGGER_GET_LOG_LEN：得到当前日志 buffer 中未被读出的日志长度。
☑ LOGGER_GET_NEXT_ENTRY_LEN：用于得到下一条日志长度。
☑ LOGGER_FLUSH_LOG：用于清空日志。

在文件 logger.h 中定义的如下所示的宏，分别对应于上述命令操作。

```
#define LOGGER_GET_LOG_BUF_SIZE_IO(__LOGGERIO, 1)
#define LOGGER_GET_LOG_LEN_IO(__LOGGERIO, 2)
#define LOGGER_GET_NEXT_ENTRY_LEN_IO(__LOGGERIO, 3)
#define LOGGER_FLUSH_LOG_IO(__LOGGERIO, 4)
```

（14）释放打开的日志设备文件：函数 logger_release()的功能是释放打开的日志设备文件，只要打开了日志就要释放。函数 logger_release()的具体实现代码如下所示。

```
static int logger_release(struct inode *ignored, struct file *file)
{
        if (file->f_mode & FMODE_READ) {
                struct logger_reader *reader = file->private_data;
                struct logger_log *log = reader->log;

                mutex_lock(&log->mutex);
                list_del(&reader->list);
                mutex_unlock(&log->mutex);

                kfree(reader);
        }

        return 0;
}
```

在上述代码中，首先判断日志是否为只读模式，如果是则直接通过 file->private_data 取得其对应的 logger_reader，然后删除其队列并释放。因为写操作没有额外分配空间，所以不需要对应的释放处理。

6.2　日志库 Liblog 驱动

在 Android 系统中，在运行库中有一个用于和 Logger 日志驱动程序进行交互的日志库 Liblog。Liblog 提供了一个向 Logger 日志驱动程序写入记录的接口，应用程序可以使用这个接口写入日志记录。在 Android 系统源码中，文件/system/core/liblog/logd_write.c 提供了日志驱动程序写入记录的接口。下面将详细分析这个文件的实现过程。

6.2.1 定义指针的初始化操作

定义函数指针 write_to_log 的初始化操作，设置在开始时为__write_to_log_init，具体代码如下所示。

```
static int __write_to_log_init(log_id_t, struct iovec *vec, size_t nr);
static int (*write_to_log)(log_id_t, struct iovec *vec, size_t nr) = __write_to_log_init;
#ifdef HAVE_PTHREADS
static pthread_mutex_t log_init_lock = PTHREAD_MUTEX_INITIALIZER;
#endif
```

当第一次调用函数指针 write_to_log 时，会执行函数__write_to_log_init()来初始化日志库 Liblog，具体实现代码如下所示。

```
static int log_fds[(int)LOG_ID_MAX] = { -1, -1, -1, -1 };
static int __write_to_log_init(log_id_t log_id, struct iovec *vec, size_t nr)
{
#ifdef HAVE_PTHREADS
    pthread_mutex_lock(&log_init_lock);
#endif
//如果发现函数指针 write_to_log 指向自己，则会调用函数 open()打开系统中的日志设备文件，并将得到的文件描
述保存在全局数组 log_fds 中
    if (write_to_log == __write_to_log_init) {
        log_fds[LOG_ID_MAIN] = log_open("/dev/"LOGGER_LOG_MAIN, O_WRONLY);
        log_fds[LOG_ID_RADIO] = log_open("/dev/"LOGGER_LOG_RADIO, O_WRONLY);
        log_fds[LOG_ID_EVENTS] = log_open("/dev/"LOGGER_LOG_EVENTS, O_WRONLY);
        log_fds[LOG_ID_SYSTEM] = log_open("/dev/"LOGGER_LOG_SYSTEM, O_WRONLY);
//如果成功，则将函数指针 write_to_log 指向函数__write_to_log_kernel
        write_to_log = __write_to_log_kernel;
//判断打开日志设备是否成功
        if (log_fds[LOG_ID_MAIN] < 0 || log_fds[LOG_ID_RADIO] < 0 ||
                log_fds[LOG_ID_EVENTS] < 0) {
            log_close(log_fds[LOG_ID_MAIN]);
            log_close(log_fds[LOG_ID_RADIO]);
            log_close(log_fds[LOG_ID_EVENTS]);
            log_fds[LOG_ID_MAIN] = -1;
            log_fds[LOG_ID_RADIO] = -1;
            log_fds[LOG_ID_EVENTS] = -1;
            write_to_log = __write_to_log_null;
        }
//如果不成功，则将 LOG_ID_SYSTEM 的值设置为 log_fds[LOG_ID_MAIN]，表示将类型为 system 和 main 的日
志记录写入日志设备文件/dev/log/main 中
        if (log_fds[LOG_ID_SYSTEM] < 0) {
            log_fds[LOG_ID_SYSTEM] = log_fds[LOG_ID_MAIN];
        }
    }

#ifdef HAVE_PTHREADS
    pthread_mutex_unlock(&log_init_lock);
#endif

    return write_to_log(log_id, vec, nr);
}
```

在上述代码中，LOG_ID_MAIN、LOG_ID_RADIO、LOG_ID_EVENTS 和 LOG_ID_SYSTEM 是 4 个枚举值，在文件/system/core/include/cutils/log.h 中定义，具体实现代码如下所示。

```
typedef enum {
    LOG_ID_MAIN = 0,
    LOG_ID_RADIO = 1,
    LOG_ID_EVENTS = 2,
    LOG_ID_SYSTEM = 3,
    LOG_ID_MAX
} log_id_t;
```

由此可见，函数__write_to_log_init()通过枚举调用的方式打开了如下日志设备。

- ☑　/dev/log/main。
- ☑　/dev/log/radio。
- ☑　/dev/log/events。

接下来分析函数__write_to_log_kernel()，其功能是根据参数 log_id 在全局数组 log_fds 中找到对应的日志设备文件描述符，并调用 log_write 宏将日志记录写入 Logger 日志驱动程序中。函数__write_to_log_kernel()的具体实现代码如下所示。

```
static int __write_to_log_kernel(log_id_t log_id, struct iovec *vec, size_t nr)
{
    ssize_t ret;
    int log_fd;

    if (/*(int)log_id >= 0 &&*/ (int)log_id < (int)LOG_ID_MAX) {
        log_fd = log_fds[(int)log_id];
    } else {
        return EBADF;
    }

    do {
        ret = log_writev(log_fd, vec, nr);
    } while (ret < 0 && errno == EINTR);

    return ret;
}
```

在函数__write_to_log_init()中，如果调用函数 open()打开系统中的日志设备失败，则函数指针 write_to_log 会指向函数__write_to_log_null()，此函数是一个空函数，没有具体功能，具体实现代码如下所示。

```
static int __write_to_log_null(log_id_t log_fd, struct iovec *vec, size_t nr)
{
    return -1;
}
```

6.2.2　记录日志

函数__android_log_write()的功能是写入类型为 main 的日志记录类型，具体实现代码如下所示。

```
int __android_log_write(int prio, const char *tag, const char *msg)
{
    struct iovec vec[3];
    log_id_t log_id = LOG_ID_MAIN;
    char tmp_tag[32];
```

```
    if (!tag)
        tag = "";

    /* XXX: This needs to go! */
    if (!strcmp(tag, "HTC_RIL") ||
        !strncmp(tag, "RIL", 3) || /* Any log tag with "RIL" as the prefix */
        !strncmp(tag, "IMS", 3) || /* Any log tag with "IMS" as the prefix */
        !strcmp(tag, "AT") ||
        !strcmp(tag, "GSM") ||
        !strcmp(tag, "STK") ||
        !strcmp(tag, "CDMA") ||
        !strcmp(tag, "PHONE") ||
        !strcmp(tag, "SMS")) {
            log_id = LOG_ID_RADIO;
            // Inform third party apps/ril/radio.. to use Rlog or RLOG
            snprintf(tmp_tag, sizeof(tmp_tag), "use-Rlog/RLOG-%s", tag);
            tag = tmp_tag;
    }

    vec[0].iov_base = (unsigned char *) &prio;
    vec[0].iov_len = 1;
    vec[1].iov_base = (void *) tag;
    vec[1].iov_len = strlen(tag) + 1;
    vec[2].iov_base = (void *) msg;
    vec[2].iov_len = strlen(msg) + 1;

    return write_to_log(log_id, vec, 3);
}
```

在上述代码中，如果传进来以 RIL 开通的标签或是 AT、GSM、STK、CDMA、PHONE、SMS 标签，则被认为是类型为 radio 的日志记录。

6.2.3 设置写入日志记录的类型

函数__android_log_buf_write 的功能是设置写入日志记录的类型，具体实现代码如下所示。

```
int __android_log_buf_write(int bufID, int prio, const char *tag, const char *msg)
{
    struct iovec vec[3];
    char tmp_tag[32];

    if (!tag)
        tag = "";

    if ((bufID != LOG_ID_RADIO) &&
          (!strcmp(tag, "HTC_RIL") ||
        !strncmp(tag, "RIL", 3) || /* Any log tag with "RIL" as the prefix */
        !strncmp(tag, "IMS", 3) || /* Any log tag with "IMS" as the prefix */
        !strcmp(tag, "AT") ||
        !strcmp(tag, "GSM") ||
        !strcmp(tag, "STK") ||
```

```
            !strcmp(tag, "CDMA") ||
            !strcmp(tag, "PHONE") ||
            !strcmp(tag, "SMS"))) {
                bufID = LOG_ID_RADIO;
                snprintf(tmp_tag, sizeof(tmp_tag), "use-Rlog/RLOG-%s", tag);
                tag = tmp_tag;
    }

    vec[0].iov_base = (unsigned char *) &prio;
    vec[0].iov_len = 1;
    vec[1].iov_base = (void *) tag;
    vec[1].iov_len = strlen(tag) + 1;
    vec[2].iov_base = (void *) msg;
    vec[2].iov_len = strlen(msg) + 1;

    return write_to_log(bufID, vec, 3);
}
```

6.2.4　向 Logger 日志驱动程序写入日志记录

函数__android_log_vprint()、__android_log_print()和__android_log_assert()的功能相同，功能是调用函数__android_log_write()以格式化字符串的形式向 Logger 日志驱动程序中写入日志记录。这 3 个函数的具体实现代码如下所示。

```
int __android_log_vprint(int prio, const char *tag, const char *fmt, va_list ap)
{
    char buf[LOG_BUF_SIZE];
    vsnprintf(buf, LOG_BUF_SIZE, fmt, ap);
    return __android_log_write(prio, tag, buf);
}
int __android_log_print(int prio, const char *tag, const char *fmt, ...)
{
    va_list ap;
    char buf[LOG_BUF_SIZE];

    va_start(ap, fmt);
    vsnprintf(buf, LOG_BUF_SIZE, fmt, ap);
    va_end(ap);

    return __android_log_write(prio, tag, buf);
}
void __android_log_assert(const char *cond, const char *tag,
                const char *fmt, ...)
{
    char buf[LOG_BUF_SIZE];

    if (fmt) {
        va_list ap;
        va_start(ap, fmt);
        vsnprintf(buf, LOG_BUF_SIZE, fmt, ap);
        va_end(ap);
```

```
    } else {
        /* Msg not provided, log condition.   N.B. Do not use cond directly as
         * format string as it could contain spurious '%' syntax (e.g.
         * "%d" in "blocks%devs == 0").
         */
        if (cond)
            snprintf(buf, LOG_BUF_SIZE, "Assertion failed: %s", cond);
        else
            strcpy(buf, "Unspecified assertion failed");
    }
    __android_log_write(ANDROID_LOG_FATAL, tag, buf);
    __builtin_trap(); /* trap so we have a chance to debug the situation */
}
```

6.2.5　记录日志数据函数

最后看记录日志数据函数__android_log_bwrite()和__android_log_btwrite()，两者写入日志记录的类型为events。这两个函数的具体实现代码如下所示。

```
int __android_log_bwrite(int32_t tag, const void *payload, size_t len)
{
    struct iovec vec[2];

    vec[0].iov_base = &tag;
    vec[0].iov_len = sizeof(tag);
    vec[1].iov_base = (void*)payload;
    vec[1].iov_len = len;

    return write_to_log(LOG_ID_EVENTS, vec, 2);
}

int __android_log_btwrite(int32_t tag, char type, const void *payload,
    size_t len)
{
    struct iovec vec[3];

    vec[0].iov_base = &tag;
    vec[0].iov_len = sizeof(tag);
    vec[1].iov_base = &type;
    vec[1].iov_len = sizeof(type);
    vec[2].iov_base = (void*)payload;
    vec[2].iov_len = len;

    return write_to_log(LOG_ID_EVENTS, vec, 3);
}
```

6.3　日志写入接口驱动

在 Android 系统中，Liblog 提供了一个向 Logger 日志驱动程序写入记录的接口，并为 C/C++和 Java 语



言提供了两种不同的 Logger 访问接口。其中 C/C++日志接口一般是在编写硬件抽象层模块或者编写 JNI 方法时使用，而 Java 接口一般是在应用层编写 APP 时使用。本节将详细讲解这两种日志写入接口驱动的基本知识。

6.3.1　C/C++层的写入接口

在 Android 系统中，通过宏来使用 C/C++层的日志接口，在文件 system/core/include/android/log.h 中定义了日志的级别，具体实现代码如下所示。

```
typedef enum android_LogPriority {
        ANDROID_LOG_UNKNOWN = 0,
        ANDROID_LOG_DEFAULT,    /* only for SetMinPriority() */
        ANDROID_LOG_VERBOSE,
        ANDROID_LOG_DEBUG,
        ANDROID_LOG_INFO,
        ANDROID_LOG_WARN,
        ANDROID_LOG_ERROR,
        ANDROID_LOG_FATAL,
        ANDROID_LOG_SILENT,/* only for SetMinPriority(); must be last */
} android_LogPriority;
```

而在文件 system/core/include/cutils/log.h 中定义了对应的宏，例如和 ANDROID_LOG_VERBOSE 对应的宏 LOGV，具体实现代码如下所示。

```
#ifndef LOG_TAG
#define LOG_TAG NULL
#endif
#ifndef LOGV
#if LOG_NDEBUG
#define LOGV(...)     ((void)0)
#else
#define LOGV(...)     ((void)LOG(LOG_VERBOSE, LOG_TAG, __VA_ARGS__))
#endif
#endif
#ifndef LOG
#define LOG(priority, tag, ...) \
        LOG_PRI(ANDROID_##priority, tag, __VA_ARGS__)
#endif
#ifndef LOG_PRI
#define LOG_PRI(priority, tag, ...) \
        android_printLog(priority, tag, __VA_ARGS__)
#endif
#define android_printLog(prio, tag, fmt...) \
        __android_log_print(prio, tag, fmt)
//下面的宏用于写入类型为 events 的日志记录
typedef enum {
      EVENT_TYPE_INT = 0,
      EVENT_TYPE_LONG = 1,
      EVENT_TYPE_STRING = 2,
      EVENT_TYPE_LIST = 3,
} AndroidEventLogType;
```

```
#ifndef LOG_EVENT_INT
#define LOG_EVENT_INT(_tag, _value) {                                   \
        int intBuf = _value;                                            \
        (void) android_btWriteLog(_tag, EVENT_TYPE_INT, &intBuf,        \
            sizeof(intBuf));                                            \
    }
#endif
#ifndef LOG_EVENT_LONG
#define LOG_EVENT_LONG(_tag, _value) {                                  \
        long long longBuf = _value;                                     \
        (void) android_btWriteLog(_tag, EVENT_TYPE_LONG, &longBuf,      \
            sizeof(longBuf));                                           \
    }
#endif
#ifndef LOG_EVENT_STRING
#define LOG_EVENT_STRING(_tag, _value)                                  \
    ((void) 0)   /* not implemented -- must combine len with string */
#endif
```

6.3.2 Java 层的写入接口

在 Android 应用程序中，通常调用应用程序框架层的 Java 接口（android.util.Log）来使用日志系统，这个 Java 接口通过 JNI 方法和系统运行库来调用内核驱动程序 Logger 把 Log 写到内核空间中。下面将详细讲解应用程序层（Application）的接口调用到内核空间的过程。

在文件 frameworks/base/core/java/android/util/Log.java 中定义了日志系统的 Java 接口，主要实现代码如下所示。

```
package android.util;
import com.android.internal.os.RuntimeInit;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.net.UnknownHostException;
public final class Log {
    public static final int VERBOSE = 2;
    public static final int DEBUG = 3;

    public static final int INFO = 4;
    public static final int WARN = 5;
    public static final int ERROR = 6;
    public static final int ASSERT = 7;
    private static class TerribleFailure extends Exception {
        TerribleFailure(String msg, Throwable cause) { super(msg, cause); }
    }
    public interface TerribleFailureHandler {
        void onTerribleFailure(String tag, TerribleFailure what);
    }

    private static TerribleFailureHandler sWtfHandler = new TerribleFailureHandler() {
            public void onTerribleFailure(String tag, TerribleFailure what) {
                RuntimeInit.wtf(tag, what);
```

```
            }
        };

    private Log() {
    }
    public static int v(String tag, String msg) {
        return println_native(LOG_ID_MAIN, VERBOSE, tag, msg);
    }
...
    public static String getStackTraceString(Throwable tr) {
        if (tr == null) {
            return "";
        }

        // This is to reduce the amount of log spew that apps do in the non-error
        // condition of the network being unavailable
        Throwable t = tr;
        while (t != null) {
            if (t instanceof UnknownHostException) {
                return "";
            }
            t = t.getCause();
        }

        StringWriter sw = new StringWriter();
        PrintWriter pw = new PrintWriter(sw);
        tr.printStackTrace(pw);
        return sw.toString();
    }
    public static int println(int priority, String tag, String msg) {
        return println_native(LOG_ID_MAIN, priority, tag, msg);
    }
            int priority, String tag, String msg);
}
```

在上述代码的结尾处，一共定义了 4 个日志缓冲区 ID。我们知道在 Logger 驱动程序模块中，定义了 log_main、log_events 和 log_radio 3 个日志缓冲区，分别对应 3 个设备文件/dev/log/main、/dev/log/events 和/dev/log/radio。在上述代码中，通过 4 个日志缓冲区的前面 3 个 ID 对应了这 3 个设备文件的文件描述符。在下载的 Android 内核源代码中，第 4 个日志缓冲区 LOG_ID_SYSTEM 并没有对应的设备文件，在这种情况下，它和 LOG_ID_MAIN 对应同一个缓冲区 ID。

其实在 Log 接口中最核心的功能是声明了 println_native 本地方法，所有的 Log 接口都是通过调用这个本地方法来实现 Log 的定义。本地方法 println_native 在文件 frameworks/base/core/jni/android_util_Log.cpp 中定义，具体实现代码如下所示。

```
#define LOG_NAMESPACE "log.tag."
#define LOG_TAG "Log_println"

#include <assert.h>
#include <cutils/properties.h>
#include <utils/Log.h>
#include <utils/String6.h>
```

```
#include "jni.h"
#include "JNIHelp.h"
#include "utils/misc.h"
#include "android_runtime/AndroidRuntime.h"
#include "android_util_Log.h"

#define MIN(a,b) ((a<b)?a:b)

namespace android {

struct levels_t {
	jint verbose;
	jint debug;
	jint info;
	jint warn;
	jint error;
	jint assert;
};
static levels_t levels;

static int toLevel(const char* value)
{
	switch (value[0]) {
		case 'V': return levels.verbose;
		case 'D': return levels.debug;
		case 'I': return levels.info;
		case 'W': return levels.warn;
		case 'E': return levels.error;
		case 'A': return levels.assert;
		case 'S': return -1; // SUPPRESS
	}
	return levels.info;
}

static jboolean isLoggable(const char* tag, jint level) {
	String8 key;
	key.append(LOG_NAMESPACE);
	key.append(tag);

	char buf[PROPERTY_VALUE_MAX];
	if (property_get(key.string(), buf, "") <= 0) {
		buf[0] = '\0';
	}

	int logLevel = toLevel(buf);
	return logLevel >= 0 && level >= logLevel;
}

static jboolean android_util_Log_isLoggable(JNIEnv* env, jobject clazz, jstring tag, jint level)
{
```

```
    if (tag == NULL) {
        return false;
    }

    const char* chars = env->GetStringUTFChars(tag, NULL);
    if (!chars) {
        return false;
    }

    jboolean result = false;
    if ((strlen(chars)+sizeof(LOG_NAMESPACE)) > PROPERTY_KEY_MAX) {
        char buf2[200];
        snprintf(buf2, sizeof(buf2), "Log tag \"%s\" exceeds limit of %d characters\n",
                chars, PROPERTY_KEY_MAX - sizeof(LOG_NAMESPACE));

        jniThrowException(env, "java/lang/IllegalArgumentException", buf2);
    } else {
        result = isLoggable(chars, level);
    }

    env->ReleaseStringUTFChars(tag, chars);
    return result;
}

bool android_util_Log_isVerboseLogEnabled(const char* tag) {
    return isLoggable(tag, levels.verbose);
}
static jint android_util_Log_println_native(JNIEnv* env, jobject clazz,
        jint bufID, jint priority, jstring tagObj, jstring msgObj)
{
    const char* tag = NULL;
    const char* msg = NULL;

    if (msgObj == NULL) {
        jniThrowNullPointerException(env, "println needs a message");
        return -1;
    }

    if (bufID < 0 || bufID >= LOG_ID_MAX) {
        jniThrowNullPointerException(env, "bad bufID");
        return -1;
    }

    if (tagObj != NULL)
        tag = env->GetStringUTFChars(tagObj, NULL);
    msg = env->GetStringUTFChars(msgObj, NULL);

    int res = __android_log_buf_write(bufID, (android_LogPriority)priority, tag, msg);

    if (tag != NULL)
        env->ReleaseStringUTFChars(tagObj, tag);
```

```
    env->ReleaseStringUTFChars(msgObj, msg);

    return res;
}

/*
 * JNI 注册
 */
static JNINativeMethod gMethods[] = {
    /* name, signature, funcPtr */
    { "isLoggable", "(Ljava/lang/String;I)Z", (void*) android_util_Log_isLoggable },
    { "println_native", "(IILjava/lang/String;Ljava/lang/String;)I", (void*) android_util_Log_println_native },
};

int register_android_util_Log(JNIEnv* env)
{
    jclass clazz = env->FindClass("android/util/Log");

    if (clazz == NULL) {
        ALOGE("Can't find android/util/Log");
        return -1;
    }

    levels.verbose = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "VERBOSE", "I"));
    levels.debug = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "DEBUG", "I"));
    levels.info = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "INFO", "I"));
    levels.warn = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "WARN", "I"));
    levels.error = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "ERROR", "I"));
    levels.assert = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "ASSERT", "I"));

    return AndroidRuntime::registerNativeMethods(env, "android/util/Log", gMethods, NELEM(gMethods));
}

};
```

通过上述代码可知，在变量 gMethods 中定义了本地方法 println_native 对应的调用函数 android_util_Log_println_native()。在函数 android_util_Log_println_native()中，如果验证的各项参数都正确，则调用运行时库函数__android_log_buf_write()来实现 Log 的写入操作。函数__android_log_buf_write 在 liblog 库中实现，拥有如下 4 个参数。

☑ 缓冲区 ID。
☑ 优先级别 ID。
☑ Tag 字符串。
☑ Msg 字符串。

接下来看文件/frameworks/base/core/java/android/util/Slog.java，功能是写入类型为 system 的日志记录，主要实现代码如下所示。

```
public final class Slog {
    private Slog() {
    }
    public static int v(String tag, String msg) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.VERBOSE, tag, msg);
```

```
    }
    public static int v(String tag, String msg, Throwable tr) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.VERBOSE, tag,
                msg + '\n' + Log.getStackTraceString(tr));
    }
    public static int d(String tag, String msg) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.DEBUG, tag, msg);
    }
    public static int d(String tag, String msg, Throwable tr) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.DEBUG, tag,
                msg + '\n' + Log.getStackTraceString(tr));
    }
    public static int i(String tag, String msg) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.INFO, tag, msg);
    }
    public static int i(String tag, String msg, Throwable tr) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.INFO, tag,
                msg + '\n' + Log.getStackTraceString(tr));
    }
    public static int w(String tag, String msg) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.WARN, tag, msg);
    }
    public static int w(String tag, String msg, Throwable tr) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.WARN, tag,
                msg + '\n' + Log.getStackTraceString(tr));
    }
    public static int w(String tag, Throwable tr) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.WARN, tag, Log.getStackTraceString(tr));
    }
    public static int e(String tag, String msg) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.ERROR, tag, msg);
    }
    public static int e(String tag, String msg, Throwable tr) {
        return Log.println_native(Log.LOG_ID_SYSTEM, Log.ERROR, tag,
                msg + '\n' + Log.getStackTraceString(tr));
    }
    public static int println(int priority, String tag, String msg) {
        return Log.println_native(Log.LOG_ID_SYSTEM, priority, tag, msg);
    }
}
```

再看文件/frameworks/base/core/java/android/util/EventLog.java，功能是提供了 4 种重载 JNI 方法向 Logger 日志驱动程序中写入类型为 events 的日志记录，记录内容有整数、长整数、字符串和列表 4 种类型。EventLog.java 的主要实现代码如下所示。

```
public static native int writeEvent(int tag, int value);
public static native int writeEvent(int tag, long value);
public static native int writeEvent(int tag, String str);
public static native int writeEvent(int tag, Object... list);
public static native void readEvents(int[] tags, Collection<Event> output)
        throws IOException;
public static String getTagName(int tag) {
```

```
        readTagsFile();
        return sTagNames.get(tag);
    }
    public static int getTagCode(String name) {
        readTagsFile();
        Integer code = sTagCodes.get(name);
        return code != null ? code : -1;
    }
    private static synchronized void readTagsFile() {
        if (sTagCodes != null && sTagNames != null) return;
        sTagCodes = new HashMap<String, Integer>();
        sTagNames = new HashMap<Integer, String>();
        Pattern comment = Pattern.compile(COMMENT_PATTERN);
        Pattern tag = Pattern.compile(TAG_PATTERN);
        BufferedReader reader = null;
        String line;
        try {
            reader = new BufferedReader(new FileReader(TAGS_FILE), 256);
            while ((line = reader.readLine()) != null) {
                if (comment.matcher(line).matches()) continue;
                Matcher m = tag.matcher(line);
                if (!m.matches()) {
                    Log.wtf(TAG, "Bad entry in " + TAGS_FILE + ": " + line);
                    continue;
                }
                try {
                    int num = Integer.parseInt(m.group(1));
                    String name = m.group(2);
                    sTagCodes.put(name, num);
                    sTagNames.put(num, name);
                } catch (NumberFormatException e) {
                    Log.wtf(TAG, "Error in " + TAGS_FILE + ": " + line, e);
                }
             }
        } catch (IOException e) {
            Log.wtf(TAG, "Error reading " + TAGS_FILE, e);
            // Leave the maps existing but unpopulated
        } finally {
            try { if (reader != null) reader.close(); } catch (IOException e) {}
        }
    }
```

再看 JNI 函数/frameworks/base/core/jni/android_util_EventLog.cpp，功能是写入整数和长整数类型的日志记录数据，主要实现代码如下所示。

```
static jint android_util_EventLog_writeEvent_Integer(JNIEnv* env, jobject clazz,
                                                     jint tag, jint value)
{
    return android_btWriteLog(tag, EVENT_TYPE_INT, &value, sizeof(value));
}
static jint android_util_EventLog_writeEvent_Long(JNIEnv* env, jobject clazz,
                                                  jint tag, jlong value)
```

```
{
    return android_btWriteLog(tag, EVENT_TYPE_LONG, &value, sizeof(value));
}
static jint android_util_EventLog_writeEvent_String(JNIEnv* env, jobject clazz,
                                                    jint tag, jstring value) {
    uint8_t buf[MAX_EVENT_PAYLOAD];
    // Don't throw NPE -- I feel like it's sort of mean for a logging function
    // to be all crashy if you pass in NULL -- but make the NULL value explicit
    const char *str = value != NULL ? env->GetStringUTFChars(value, NULL) : "NULL";
    uint32_t len = strlen(str);
    size_t max = sizeof(buf) - sizeof(len) - 2;   // Type byte, final newline
    if (len > max) len = max;
    buf[0] = EVENT_TYPE_STRING;
    memcpy(&buf[1], &len, sizeof(len));
    memcpy(&buf[1 + sizeof(len)], str, len);
    buf[1 + sizeof(len) + len] = '\n';
    if (value != NULL) env->ReleaseStringUTFChars(value, str);
    return android_bWriteLog(tag, buf, 2 + sizeof(len) + len);
}
```

在上述代码中，通过调用宏 android_btWriteLog 的方式向日志驱动程序中写入日志记录数据，并且还定义了 JNI 方法 writeEvent 的具体实现方法 android_util_EventLog_writeEvent_String()，方法 writeEvent 的功能是写入字符串类型的日志记录。

再看文件 android_util_EventLog.cpp 中的函数，此函数是本地 JNI 函数 writeEvent(long tag, Object... value)的具体实现，具体实现代码如下所示。

```
static jint android_util_EventLog_writeEvent_Array(JNIEnv* env, jobject clazz,
                                                   jint tag, jobjectArray value) {
    if (value == NULL) {
        return android_util_EventLog_writeEvent_String(env, clazz, tag, NULL);
    }

    uint8_t buf[MAX_EVENT_PAYLOAD];
    const size_t max = sizeof(buf) - 1;   // leave room for final newline
    size_t pos = 2;   // Save room for type tag & array count

    jsize copied = 0, num = env->GetArrayLength(value);
    for (; copied < num && copied < 255; ++copied) {
        jobject item = env->GetObjectArrayElement(value, copied);
        if (item == NULL || env->IsInstanceOf(item, gStringClass)) {
            if (pos + 1 + sizeof(jint) > max) break;
            const char *str = item != NULL ? env->GetStringUTFChars((jstring) item, NULL) : "NULL";
            jint len = strlen(str);
            if (pos + 1 + sizeof(len) + len > max) len = max - pos - 1 - sizeof(len);
            buf[pos++] = EVENT_TYPE_STRING;
            memcpy(&buf[pos], &len, sizeof(len));
            memcpy(&buf[pos + sizeof(len)], str, len);
            pos += sizeof(len) + len;
            if (item != NULL) env->ReleaseStringUTFChars((jstring) item, str);
        } else if (env->IsInstanceOf(item, gIntegerClass)) {
            jint intVal = env->GetIntField(item, gIntegerValueID);
            if (pos + 1 + sizeof(intVal) > max) break;
```

```
            buf[pos++] = EVENT_TYPE_INT;
            memcpy(&buf[pos], &intVal, sizeof(intVal));
            pos += sizeof(intVal);
        } else if (env->IsInstanceOf(item, gLongClass)) {
            jlong longVal = env->GetLongField(item, gLongValueID);
            if (pos + 1 + sizeof(longVal) > max) break;
            buf[pos++] = EVENT_TYPE_LONG;
            memcpy(&buf[pos], &longVal, sizeof(longVal));
            pos += sizeof(longVal);
        } else {
            jniThrowException(env,
                    "java/lang/IllegalArgumentException",
                    "Invalid payload item type");
            return -1;
        }
        env->DeleteLocalRef(item);
    }

    buf[0] = EVENT_TYPE_LIST;
    buf[1] = copied;
    buf[pos++] = '\n';
    return android_bWriteLog(tag, buf, pos);
}
```

第 7 章　Ashmem 驱动详解

Android 系统中提供了独特的匿名共享内存子系统 Ashmem（Anonymous Shared Memory），它以驱动程序的形式实现在内核空间中。Ashmem 有如下两个特点：

☑　能够辅助内存管理系统来有效地管理不再使用的内存块。

☑　通过 Binder 进程间通信机制实现进程间的内存共享。

Android 系统的匿名共享内存 Ashmem 驱动程序利用了 Linux 的共享内存子系统导出的接口来实现自己的功能。在 Android 系统匿名共享内存系统中，其核心功能是实现创建（open）、映射（mmap）、读写（read/write）以及锁定和解锁（pin/unpin）。本章将详细讲解 Android 内存驱动 Ashmem 的基本架构知识。

7.1　分析 Ashmem 驱动程序

Android 系统的匿名共享内存子系统的主体是以驱动程序的形式实现在内核空间的，同时在系统运行时库层和应用程序框架层提供了访问接口。其中在系统运行时库层中提供了 C/C++调用接口，而在应用程序框架层提供了 Java 调用接口，本节将详细讲解 Ashmem 驱动程序的基本知识。

7.1.1　基础数据结构

在 Ashmem 驱动程序中，用到了 ashmem_area、ashmem_range 和 ashmem_pin 个结构体。其中前两个结构体在文件 kernel/goldfish/mm/ashmem.c 中定义，具体实现代码如下所示。

```
struct ashmem_area {
        char name[ASHMEM_FULL_NAME_LEN];        /*匿名共享内存的名称*/
        struct list_head unpinned_list;         /*解锁内存列表*/
        struct file *file;                      /*指向临时文件系统 tmpfs 中的一个文件*/
        size_t size;                            /*文件大小*/
        unsigned long prot_mask;                /*匿名共享内存的访问保护位*/
};
struct ashmem_range {
        struct list_head lru;                   /*最近最少使用的列表*/
        struct list_head unpinned;              /*entry in its area's unpinned list*/
        struct ashmem_area *asma;               /*associated area*/
        size_t pgstart;                         /*处于解锁状态内存的开始地址*/
        size_t pgend;                           /*处于解锁状态内存的结束地址*/
        unsigned int purged;                    /*解锁内存是否被收回*/
};
```

结构体 ashmem_area 用于表示一块匿名共享内存单元，结构体 ashmem_range 用于表示处于解锁状态的内存。

结构体 ashmem_pin 用于表示被锁定或被解锁的内存，在文件 kernel/goldfish/include/linux/ashmem.h 中定义，

具体代码如下所示。

```
struct ashmem_pin {
		__u32 offset;				/*这块内存的偏移值*/
		__u32 len;					/*这块内存的大小 */
};
```

结构体 ashmem_fops 定义了 dev/ashmem 的操作方法列表，具体代码如下所示。

```
static struct file_operations ashmem_fops = {
		.owner = THIS_MODULE,
		.open = ashmem_open,
		.release = ashmem_release,
		.mmap = ashmem_mmap,
		.unlocked_ioctl = ashmem_ioctl,
		.compat_ioctl = ashmem_ioctl,
};
```

7.1.2 驱动初始化

在 Android 系统中，通过 Ashmem 驱动初始化函数可以获取如下两点信息。

☑ Ashmem 给用户空间暴露了什么接口，即创建了什么样的设备文件。

☑ Ashmem 提供了什么函数来操作这个设备文件。

Ashmem 驱动程序在文件 kernel/common/mm/ashmem.c 中实现，其中函数 ashmem_init()实现模块初始化处理，主要实现代码如下所示。

```
static struct miscdevice ashmem_misc = {
	.minor = MISC_DYNAMIC_MINOR,
	.name = "ashmem",
	.fops = &ashmem_fops,
};
static int __init ashmem_init(void)
{
	int ret;
	...
	ret = misc_register(&ashmem_misc);
	if (unlikely(ret)) {
		printk(KERN_ERR "ashmem: failed to register misc device!\n");
		return ret;
	}
	...
	return 0;
}
```

在上述代码中，在加载 Ashmem 驱动程序时会创建一个设备文件/dev/ashmem，这是一个 misc 类型的设备。通过函数 misc_register()来注册 misc 设备，调用这个函数后会在/dev 目录下生成一个 ashmem 设备文件。在设备文件中一共提供了 open、mmap、release 和 ioctl 4 种操作，此处并没有 read 和 write 操作，原因是读写共享内存的方法是通过内存映射地址来进行的，通过 mmap 系统调用将这个设备文件映射到进程地址空间中。与此同时，直接对内存进行了读写操作，所以不需要通过 read 和 write 方式进行文件操作。

匿名共享内存创建功能是在文件 frameworks/base/core/java/android/os/MemoryFile.java 中实现的，此文件调用了类 MemoryFile 的构造函数，MemoryFile 的构造函数调用了 JNI 函数 native_open()，这样便创建了匿名内存共享文件。JNI 方法 native_open()在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中实现，具体代

码如下所示。

```
static jobject android_os_MemoryFile_open(JNIEnv* env, jobject clazz, jstring name, jint length)
{
        const char* namestr = (name ? env->GetStringUTFChars(name, NULL) : NULL);
        int result = ashmem_create_region(namestr, length);
        if (name)
            env->ReleaseStringUTFChars(name, namestr);
        if (result < 0) {
            jniThrowException(env, "java/io/IOException", "ashmem_create_region failed");
            return NULL;
        }
        return jniCreateFileDescriptor(env, result);
}
```

函数 native_open()通过运行时库提供的接口 ashmem_create_region 创建匿名共享内存，这个接口在文件 system/core/libcutils/ashmem-dev.c 中实现，具体代码如下所示。

```
int ashmem_create_region(const char *name, size_t size)
{
     int fd, ret;
     fd = open(ASHMEM_DEVICE, O_RDWR);
     if (fd < 0)
          return fd;
     if (name) {
          char buf[ASHMEM_NAME_LEN];
          strlcpy(buf, name, sizeof(buf));
          ret = ioctl(fd, ASHMEM_SET_NAME, buf);
          if (ret < 0)
               goto error;
     }
     ret = ioctl(fd, ASHMEM_SET_SIZE, size);
     if (ret < 0)
          goto error;
     return fd;
error:
     close(fd);
     return ret;
}
```

在上述代码中，通过执行 3 个文件操作系统调用的方式和 Ashmem 驱动程序进行交互。通过 open 操作打开设备文件 ASHMEM_DEVICE，通过 ioctl 操作设置匿名共享内存的名称和大小。

7.1.3　打开匿名共享内存设备文件

进入内核后，open 会调用函数 ashmem_open()打开匿名共享内存设备文件。函数 ashmem_op()能够为程序创建一个 ashmem_area 结构体，具体实现代码如下所示。

```
static int ashmem_open(struct inode *inode, struct file *file)
{
     struct ashmem_area *asma;
     int ret;
     ret = nonseekable_open(inode, file);
     if (unlikely(ret))
```

```
        return ret;
    asma = kmem_cache_zalloc(ashmem_area_cachep, GFP_KERNEL);
    if (unlikely(!asma))
        return -ENOMEM;
    INIT_LIST_HEAD(&asma->unpinned_list);
    memcpy(asma->name, ASHMEM_NAME_PREFIX, ASHMEM_NAME_PREFIX_LEN);
    asma->prot_mask = PROT_MASK;
    file->private_data = asma;
    return 0;
}
```

上述代码的执行流程如下。

- ☑ 通过函数 nonseekable_open()设置这个文件不可以执行定位操作，即不可执行 seek 文件操作。
- ☑ 通过函数 kmem_cache_zalloc()在刚创建的 slab 缓冲区 ashmem_area_cachep 中创建一个 ashmem_area 结构体，并将创建的结构体保存在本地变量 asma 中。
- ☑ 初始化变量 asma 的其他域，其中域 name 初始为宏 ASHMEM_NAME_PREFIX，宏 ASHMEM_NAME_PREFIX 的定义代码如下。

```
#define ASHMEM_NAME_PREFIX "dev/ashmem/"
#define ASHMEM_NAME_PREFIX_LEN (sizeof(ASHMEM_NAME_PREFIX) - 1)
```

- ☑ 将结构 ashmem_area 保存在打开文件结构体的 private_data 域中，此时通过使用 Ashmem 驱动程序，可以在其他模块通过 private_data 域取回这个 ashmem_area 结构。

在函数 ashmem_create_region()中调用了两次 ioctl 文件操作，功能是设置新建匿名共享内存的名字和大小。在文件 kernel/comon/mm/include/ashmem.h 中，ASHMEM_SET_NAME 和 ASHMEM_SET_SIZE 分别表示新建内存的名字和大小，具体定义代码如下所示。

```
#define ASHMEM_NAME_LEN        256
#define __ASHMEMIOC            0x77
#define ASHMEM_SET_NAME        _IOW(__ASHMEMIOC, 1, char[ASHMEM_NAME_LEN])
#define ASHMEM_SET_SIZE        _IOW(__ASHMEMIOC, 3, size_t)
```

其中 ASHMEM_SET_NAME 的 ioctl 调用会进入 Ashmem 驱动程序函数 ashmem_ioctl()中，此函数能够将从用户空间传进来的匿名共享内存的大小值保存在对应的 asma->size 域中。函数 ashmem_ioctl()的实现代码如下所示。

```
static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        struct ashmem_area *asma = file->private_data;
        long ret = -ENOTTY;
        switch (cmd) {
        case ASHMEM_SET_NAME:
                ret = set_name(asma, (void __user *) arg);
                break;
        case ASHMEM_GET_NAME:
                ret = get_name(asma, (void __user *) arg);
                break;
        case ASHMEM_SET_SIZE:
                ret = -EINVAL;
                if (!asma->file) {
                        ret = 0;
                        asma->size = (size_t) arg;
                }
```

```
			break;
		case ASHMEM_GET_SIZE:
			ret = asma->size;
			break;
		case ASHMEM_SET_PROT_MASK:
			ret = set_prot_mask(asma, arg);
			break;
		case ASHMEM_GET_PROT_MASK:
			ret = asma->prot_mask;
			break;
		case ASHMEM_PIN:
		case ASHMEM_UNPIN:
		case ASHMEM_GET_PIN_STATUS:
			ret = ashmem_pin_unpin(asma, cmd, (void __user *) arg);
			break;
		case ASHMEM_PURGE_ALL_CACHES:
			ret = -EPERM;
			if (capable(CAP_SYS_ADMIN)) {
				ret = ashmem_shrink(0, GFP_KERNEL);
				ashmem_shrink(ret, GFP_KERNEL);
			}
			break;
	}
	return ret;
}
```

上述代码主要完成如下两个功能。

☑　struct ashmem_area *asma = file->private_data：获取描述将要改名的匿名共享内存 asma。

☑　ret = set_name(asma, (void __user *) arg)：调用函数 set_name 修改匿名共享内存的名称。

函数 set_name()也是在文件 kernel/goldfish/mm/ashmem.c 中实现的，功能是把用户空间传进来的匿名共享内存的名字设置到 asma->name 域中。函数 set_name()的具体实现代码如下所示。

```
static int set_name(struct ashmem_area *asma, void __user *name)
{
	int ret = 0;
	mutex_lock(&ashmem_mutex);
	/* cannot change an existing mapping's name */
	if (unlikely(asma->file)) {
		ret = -EINVAL;
		goto out;
	}
	if (unlikely(copy_from_user(asma->name + ASHMEM_NAME_PREFIX_LEN,
						name, ASHMEM_NAME_LEN)))
		ret = -EFAULT;
	asma->name[ASHMEM_FULL_NAME_LEN-1] = '\0';
out:
	mutex_unlock(&ashmem_mutex);
	return ret;
}
```

到此为止，创建匿名共享内存的过程就全部介绍完毕了。

7.1.4 内存映射

Ashmem 驱动程序并不提供文件的 read 操作和 write 操作，如果进程要访问这个共享内存，则必须将这个设备文件映射到自己的进程空间中，然后才能进行内存访问。在类 MemoryFile 的构造函数中，创建匿名共享内存后需要把匿名共享内存设备文件映射到进程空间。映射功能是通过调用 JNI 方法 native_mmap()实现的，此 JNI 方法在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中实现，具体实现代码如下所示。

```
static jint android_os_MemoryFile_mmap(JNIEnv* env, jobject clazz, jobject fileDescriptor,
        jint length, jint prot)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    jint result = (jint)mmap(NULL, length, prot, MAP_SHARED, fd, 0);
    if (!result)
        jniThrowException(env, "java/io/IOException", "mmap failed");
    return result;
}
```

在上述代码中，在 open 匿名设备文件/dev/ashmem 获得文件描述符 fd。有了这个文件描述符后，就可以直接通过函数 mmap()执行内存映射操作了。当调用函数 mmap()打开映射到进程的地址空间时，会立即执行 ashmem 中的函数 ashmem_mmap()。函数 ashmem_mmap()的功能是，调用 Linux 内核中的函数 shmem_file_setup()在临时文件系统 tmpfs 中创建一个临时文件，这个临时文件与 Ashmem 驱动程序创建的匿名共享内存对应。函数 ashmem_mmap()在文件 kernel/goldfish/mm/ashmem.c 中定义，具体实现代码如下所示。

```
static int ashmem_mmap(struct file *file, struct vm_area_struct *vma)
{
        struct ashmem_area *asma = file->private_data;
        int ret = 0;
        mutex_lock(&ashmem_mutex);
        /* user needs to SET_SIZE before mapping */
        if (unlikely(!asma->size)) {
                ret = -EINVAL;
                goto out;
        }
        /* requested protection bits must match our allowed protection mask */
        if (unlikely((vma->vm_flags & ~asma->prot_mask) & PROT_MASK)) {
                ret = -EPERM;
                goto out;
        }
        if (!asma->file) {
                char *name = ASHMEM_NAME_DEF;
                struct file *vmfile;
                if (asma->name[ASHMEM_NAME_PREFIX_LEN] != '\0')
                        name = asma->name;
                /* ... and allocate the backing shmem file */
                vmfile = shmem_file_setup(name, asma->size, vma->vm_flags);
                if (unlikely(IS_ERR(vmfile))) {
                        ret = PTR_ERR(vmfile);
                        goto out;
                }
                asma->file = vmfile;
```

```
		}
		get_file(asma->file);
		if (vma->vm_flags & VM_SHARED)
			shmem_set_file(vma, asma->file);
		else {
			if (vma->vm_file)
				fput(vma->vm_file);
			vma->vm_file = asma->file;
		}
		vma->vm_flags |= VM_CAN_NONLINEAR;
out:
		mutex_unlock(&ashmem_mutex);
		return ret;
}
```

在上述代码中，检查了虚拟内存 vma 是否允许在不同进程之间实现共享。如果允许则调用函数 shmem_set_file()来设置它的映射文件和内存操作方法表。

7.1.5　读写操作

从类 MemoryFile 中可以获得读写操作的过程，对应的代码如下所示。

```
private static native int native_read(FileDescriptor fd, int address, byte[] buffer,
int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
private static native void native_write(FileDescriptor fd, int address, byte[] buffer,
int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
private FileDescriptor mFD;            // ashmem file descriptor
private int mAddress;    // address of ashmem memory
private int mLength;     // total length of our ashmem region
private boolean mAllowPurging = false;   // true if our ashmem region is unpinned
public int readBytes(byte[] buffer, int srcOffset, int destOffset, int count)
    throws IOException {
        if (isDeactivated()) {
            throw new IOException("Can't read from deactivated memory file.");
        }
        if (destOffset < 0 || destOffset > buffer.length || count < 0
            || count > buffer.length - destOffset
            || srcOffset < 0 || srcOffset > mLength
            || count > mLength - srcOffset) {
                throw new IndexOutOfBoundsException();
        }
        return native_read(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
    }
    public void writeBytes(byte[] buffer, int srcOffset, int destOffset, int count)
        throws IOException {
            if (isDeactivated()) {
                throw new IOException("Can't write to deactivated memory file.");
            }
            if (srcOffset < 0 || srcOffset > buffer.length || count < 0
                || count > buffer.length - srcOffset
                || destOffset < 0 || destOffset > mLength
                || count > mLength - destOffset) {
```

```
                    throw new IndexOutOfBoundsException();
            }
            native_write(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
```

通过对上述代码的分析可知，是通过调用 JNI 方法实现读写匿名共享内存操作功能。读操作的 JNI 方法是 native_read()，写操作的 NI 方法是 native_write()，这两个方法都在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jint android_os_MemoryFile_read(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }

    env->SetByteArrayRegion(buffer, destOffset, count, (const jbyte *)address + srcOffset);

    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}

static jint android_os_MemoryFile_write(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }
    env->GetByteArrayRegion(buffer, srcOffset, count, (jbyte *)address + destOffset);
    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}
```

在上述代码中，函数 ashmem_pin_region()和函数 ashmem_unpin_region()用于为系统运行时库提供接口，功能是执行匿名共享内存的锁定和解锁操作。这样就能够通知 Ashmem 驱动程序哪些内存块是正在使用的，哪些需要锁定，哪些不需要使用，哪些可以解锁。这两个函数在文件 system/core/libcutils/ashmem-dev.c 中定义，具体实现代码如下所示。

```
int ashmem_pin_region(int fd, size_t offset, size_t len)
{
     struct ashmem_pin pin = { offset, len };
```

```
	return ioctl(fd, ASHMEM_PIN, &pin);
}
int ashmem_unpin_region(int fd, size_t offset, size_t len)
{
	struct ashmem_pin pin = { offset, len };
	return ioctl(fd, ASHMEM_UNPIN, &pin);
}
```

经过上述操作之后，Ashmem 驱动程序就可以在整个内存管理系统中管理内存了。

7.1.6　锁定和解锁

在 Android 系统中，通过如下两个 ioctl 操作实现匿名共享内存的锁定和解锁操作。

☑　ASHMEM_PIN。

☑　ASHMEM_UNPIN。

ASHMEM_PIN 和 ASHMEM_UNPIN 在文件 kernel/common/include/linux/ashmem.h 中定义，对应代码如下所示。

```
#define __ASHMEMIOC		0x77
#define ASHMEM_PIN		_IOW(__ASHMEMIOC, 7, struct ashmem_pin)
#define ASHMEM_UNPIN		_IOW(__ASHMEMIOC, 8, struct ashmem_pin)
struct ashmem_pin {
	__u32 offset;	/* offset into region, in bytes, page-aligned */
	__u32 len;	/* length forward from offset, in bytes, page-aligned */
};
```

再看函数 ashmem_ioctl()，在其实现代码中和 ASHMEM_PIN 与 ASHMEM_UNPIN 这两个操作相关的代码如下所示。

```
static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
	struct ashmem_area *asma = file->private_data;
	long ret = -ENOTTY;
	switch (cmd) {
	...
	case ASHMEM_PIN:
	case ASHMEM_UNPIN:
		ret = ashmem_pin_unpin(asma, cmd, (void __user *) arg);
		break;
	...
	}
	return ret;
}
```

在上述代码中，调用函数 ashmem_pin_unpin()处理控制命令 ASHMEM_PIN 和 ASHMEM_UNPIN。函数 ashmem_pin_unpin()的实现流程如下所示。

☑　获取传递到用户空间的参数，并将获取值保存在本地变量 pin 中。这是一个 struct ashmem_pin 类型的结构体类型，在里面包括了要 pin/unpin 的内存块的起始地址和大小。

☑　因为起始地址和大小的单位都是字节，所以通过转换处理为以页面为单位的并保存在本地变量 pgstart 和 pgend 中。

☑　不但对参数进行安全性检查，还要确保只要从用户空间传进来的内存块的大小值为 0，就认为是要 pin/unpin 整个匿名共享内存。

☑ 判断当前要执行操作的类别，根据 ASHMEM_PIN 操作和 ASHMEM_UNPIN 操作分别执行 ashmem_pin 和 ashmem_unpin。

☑ 当创建匿名共享内存时，所有默认的内存都是 pinned 状态的，只有用户告诉 Ashmem 驱动程序要 unpin 某一块内存时，Ashmem 驱动程序才会把这块内存 unpin。

☑ 用户告知 Ashmem 驱动程序重新 pin 某一块前面被 unpin 过的内存块，这样能够将此内存从 unpinned 状态转换为 pinned 状态。

函数 ashmem_pin_unpin()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int ashmem_pin_unpin(struct ashmem_area *asma, unsigned long cmd,
                void __user *p)
{
        struct ashmem_pin pin;
        size_t pgstart, pgend;
        int ret = -EINVAL;

        if (unlikely(!asma->file))
                return -EINVAL;

        if (unlikely(copy_from_user(&pin, p, sizeof(pin))))
                return -EFAULT;

        /* per custom, you can pass zero for len to mean "everything onward" */
        if (!pin.len)
                pin.len = PAGE_ALIGN(asma->size) - pin.offset;

        if (unlikely((pin.offset | pin.len) & ~PAGE_MASK))
                return -EINVAL;

        if (unlikely(((__u32) -1) - pin.offset < pin.len))
                return -EINVAL;

        if (unlikely(PAGE_ALIGN(asma->size) < pin.offset + pin.len))
                return -EINVAL;

        pgstart = pin.offset / PAGE_SIZE;
        pgend = pgstart + (pin.len / PAGE_SIZE) - 1;

        mutex_lock(&ashmem_mutex);

        switch (cmd) {
        case ASHMEM_PIN:
                ret = ashmem_pin(asma, pgstart, pgend);
                break;
        case ASHMEM_UNPIN:
                ret = ashmem_unpin(asma, pgstart, pgend);
                break;
        case ASHMEM_GET_PIN_STATUS:
                ret = ashmem_get_pin_status(asma, pgstart, pgend);
                break;
        }
```

```
	mutex_unlock(&ashmem_mutex);

	return ret;
}
```

由此可见，执行 ASHMEM_PIN 操作的目标对象必须是一块处于 unpinned 状态的内存块。

函数 ashmem_unpin()的功能是解锁某一块匿名共享内存，具体处理流程如下。

☑ 在遍历 asma->unpinned_list 列表时，查找当前处于 unpinned 状态的内存块是否与将要 unpin 的内存块[pgstart, pgend]相交，如果相交则通过执行合并操作调整 pgstart 和 pgend 的大小。

☑ 调用函数 range_del()删除原来已经被 unpinned 过的内存块。

☑ 调用函数 range_alloc()重新 unpinned 调整过后的内存块[pgstart, pgend]，此时新的内存块[pgstart, pgend]已经包含了刚才所有被删除的 unpinned 状态的内存。

☑ 如果找到相交的内存块，并且调整了 pgstart 和 pgend 的大小之后，需要重新扫描 asma->unpinned_list 列表。原因是新的内存块[pgstart, pgend]可能与前后处于 unpinned 状态的内存块发生相交。

函数 ashmem_unpin()在文件 kernel/goldfish/ashmem.c 中定义，具体的实现代码如下所示。

```
static int ashmem_unpin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
{
	struct ashmem_range *range, *next;
	unsigned int purged = ASHMEM_NOT_PURGED;

restart:
	list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned) {
		/* short circuit: this is our insertion point */
		if (range_before_page(range, pgstart))
			break;

		/*
		 * The user can ask us to unpin pages that are already entirely
		 * or partially pinned. We handle those two cases here.
		 */
		if (page_range_subsumed_by_range(range, pgstart, pgend))
			return 0;
		if (page_range_in_range(range, pgstart, pgend)) {
			pgstart = min_t(size_t, range->pgstart, pgstart),
			pgend = max_t(size_t, range->pgend, pgend);
			purged |= range->purged;
			range_del(range);
			goto restart;
		}
	}
	return range_alloc(asma, range, purged, pgstart, pgend);
}
```

range_before_page 的操作是一个宏定义，功能是判断 range 描述的内存块是否在 page 页面之前，如果是则表示结束整个描述。asma->unpinned_list 列表是按照页面号从大到小进行排列的，并且每一块被 unpin 的内存都是不相交的。range_before_pag 的定义代码如下所示。

```
#define range_before_page(range, page) \
  ((range)->pgend < (page))
```

page_range_subsumed_by_range 的操作也是一个宏定义，功能是判断内存块是不是包含了[start, end]这个

内存块，如果包含则说明当前要 unpin 的内存块已经处于 unpinned 状态。如果什么也不用操作则直接返回。page_range_subsumed_by_range 的定义代码如下所示。

```
#define page_range_subsumed_by_range(range, start, end) \
  (((range)->pgstart <= (start)) && ((range)->pgend >= (end)))
```

page_range_in_range 的操作也是一个宏定义，功能是判断内存块[start,end]是否互相包含或者相交。page_range_in_range 的定义代码如下所示。

```
#define page_range_in_range(range, start, end) \
  (page_in_range(range, start) || page_in_range(range, end) || \
   page_range_subsumes_range(range, start, end))
```

page_range_subsumed_by_range 的操作也是一个宏定义，功能是判断内存块 range 是否包含内存块[start, end]。page_range_subsumed_by_range 的定义代码如下所示。

```
#define page_range_subsumed_by_range(range, start, end) \
  (((range)->pgstart <= (start)) && ((range)->pgend >= (end)))
```

range_in_range 的操作也是一个宏定义，功能是判断内存块地址 page 是否包含在内存块 range 中。range_in_range 的定义代码如下所示。

```
#define page_in_range(range, page) \
 (((range)->pgstart <= (page)) && ((range)->pgend >= (page)))
```

函数 range_del()的功能是从 asma->unpinned_list 中删除内存块，并判断它是否在 lru 列表中。函数 range_del()的具体实现代码如下所示。

```
static void range_del(struct ashmem_range *range)
{
	list_del(&range->unpinned);
	if (range_on_lru(range))
		lru_del(range);
	kmem_cache_free(ashmem_range_cachep, range);
}
```

再看函数 lru_del()，内存块的状态 purged 值为 ASHMEM_NOT_PURGED，表示现在没有收回对应的物理页面，那么内存块就位于 lru 列表中，则使用函数 lru_del()删除这个内存块。函数 lru_del()的具体实现代码如下所示。

```
static inline void lru_del(struct ashmem_range *range)
{
	list_del(&range->lru);
	lru_count -= range_size(range);
}
```

在函数 ashmem_unpin()中调用的 range_alloc()函数，其功能是从 slab 缓冲区中 ashmem_range_cachep 分配一个 ashmem_range，并进行相应的初始化处理。然后放在对应的列表 ashmem_area-> unpinned_list 中，并判断这个 range 的 purged 是否处于 ASHMEM_NOT_PURGED 状态，如果是则要把它放在 lru 列表中。函数 range_alloc()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int range_alloc(struct ashmem_area *asma,
			struct ashmem_range *prev_range, unsigned int purged,
			size_t start, size_t end)
{
    struct ashmem_range *range;
    range = kmem_cache_zalloc(ashmem_range_cachep, GFP_KERNEL);
    if (unlikely(!range))
        return -ENOMEM;
    range->asma = asma;
```

```
	range->pgstart = start;
	range->pgend = end;
	range->purged = purged;
	list_add_tail(&range->unpinned, &prev_range->unpinned);
	if (range_on_lru(range))
		lru_add(range);
	return 0;
}
```

函数 lru_add()的功能是将未被回收的已解锁内存块添加到全局列表 ashmem_lru_list 中。函数 lru_add()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static inline void lru_add(struct ashmem_range *range)
{
	list_add_tail(&range->lru, &ashmem_lru_list);
	lru_count += range_size(range);
}
```

函数 ashmem_pin()的功能是锁定一块匿名共享内存区域。被 pin 的内存块肯定被保存在 unpinned_list 列表中，如果不在则什么都不用做。要想判断在 unpinned_list 列表中是否存在 pin 的内存块，需要通过遍历 asma->unpinned_list 列表的方式找出与之相交的内存块。函数 ashmem_pin()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int ashmem_pin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
{
	struct ashmem_range *range, *next;
	int ret = ASHMEM_NOT_PURGED;

	list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned) {
		/* moved past last applicable page; we can short circuit */
		if (range_before_page(range, pgstart))
			break;
		if (page_range_in_range(range, pgstart, pgend)) {
			ret |= range->purged;

			/* Case #1: Easy. Just nuke the whole thing. */
			if (page_range_subsumes_range(range, pgstart, pgend)) {
				range_del(range);
				continue;
			}

			/* Case #2: We overlap from the start, so adjust it */
			if (range->pgstart >= pgstart) {
				range_shrink(range, pgend + 1, range->pgend);
				continue;
			}

			/* Case #3: We overlap from the rear, so adjust it */
			if (range->pgend <= pgend) {
				range_shrink(range, range->pgstart, pgstart-1);
				continue;
			}
```

```
                /*
                 * Case #4: We eat a chunk out of the middle. A bit
                 * more complicated, we allocate a new range for the
                 * second half and adjust the first chunk's endpoint.
                 */
                range_alloc(asma, range, range->purged,
                            pgend + 1, range->pgend);
                range_shrink(range, range->pgstart, pgstart - 1);
                break;
            }
        }
        return ret;
}
```

在上述代码中对重新锁定内存块操作实现了判断，通过 if 语句处理了如下 4 种情形。

☑ 指定要锁定的内存块[start,end]包含了解锁状态的内存块range，此时只要将解锁状态的内存块range从其宿主匿名共享内存的解锁内存块列表 unpinned_list 中删除即可。

☑ 合并要锁定内存块[pgstart,pgend]后部分和解锁状态内存块 range 的前半部分，此时将解锁状态内存块 range 的开始地址设置为要锁定内存块的末尾地址的下一个页面地址。

☑ 合并要锁定内存块[pgstart,pgend]前部分和解锁状态内存块 range 的后半部分，此时将解锁状态内存块 range 的末尾地址设置为要锁定内存块的开始地址的下一个页面地址。

☑ 设置要锁定内存块[pgstart,pgend]包含在解锁状态内存块 range 中。

再看函数 range_shrink()，功能是设置 range 描述的内存块的起始页面号，如果还存在于 lru 列表中，则需要调整在 lru 列表中的总页面数大小。函数 range_shrink()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static inline void range_shrink(struct ashmem_range *range,
                        size_t start, size_t end)
{
        size_t pre = range_size(range);

        range->pgstart = start;
        range->pgend = end;

        if (range_on_lru(range))
                lru_count -= pre - range_size(range);
}
```

7.1.7 回收内存块

首先分析 Ashmem 驱动初始化函数 ashmem_init()，此函数会调用函数 register_shrinker()向内存管理系统注册一个内存回收算法函数，具体实现代码如下所示。

```
static struct shrinker ashmem_shrinker = {
        .shrink = ashmem_shrink,
        .seeks = DEFAULT_SEEKS * 4,
};
static int __init ashmem_init(void)
{
        ...
```

```
        register_shrinker(&ashmem_shrinker);
        printk(KERN_INFO "ashmem: initialized\n");
        return 0;
}
```

其实在 Linux 内核程序中，当系统内存不够用时，内存管理系统就会通过调用内存回收算法的方式删除最近没有用过的内存，将这些内存从物理内存中清除，这样可以增加物理内存的容量。所以在 Android 系统中也借用了这种机制，当内存管理系统回收内存时会调用函数 ashmem_shrink()以执行内存回收操作。函数 ashmem_shrink()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int ashmem_shrink(struct shrinker *s, struct shrink_control *sc)
{
        struct ashmem_range *range, *next;
        /* We might recurse into filesystem code, so bail out if necessary */
        if (sc->nr_to_scan && !(sc->gfp_mask & __GFP_FS))
                return -1;
        if (!sc->nr_to_scan)
                return lru_count;
        mutex_lock(&ashmem_mutex);
        list_for_each_entry_safe(range, next, &ashmem_lru_list, lru) {
                loff_t start = range->pgstart * PAGE_SIZE;
                loff_t end = (range->pgend + 1) * PAGE_SIZE;
                do_fallocate(range->asma->file,
                                FALLOC_FL_PUNCH_HOLE | FALLOC_FL_KEEP_SIZE,
                                start, end - start);
                range->purged = ASHMEM_WAS_PURGED;
                lru_del(range);
                sc->nr_to_scan -= range_size(range);
                if (sc->nr_to_scan <= 0)
                        break;
        }
        mutex_unlock(&ashmem_mutex);
        return lru_count;
}
```

7.2　C++访问接口层

如果想在 Android 进程之间共享一个完整的匿名共享内存块，可以通过调用接口 MemoryHeapBase 来实现。如果只是想在进程之间共享匿名共享内存块中的一部分时，可以通过调用接口 MemoryBase 来实现。本节将分析 C++访问接口层的基本知识。

7.2.1　接口 MemoryHeapBase 的服务器端实现

接口 MemoryBase 以接口 MemoryHeapBase 为基础，这两个接口都可以作为一个 Binder 对象在进程之间进行传输。因为接口 MemoryHeapBase 是一个 Binder 对象，所以拥有 Server 端对象（必须实现一个 BnInterface 接口）和 Client 端引用（必须要实现一个 BpInterface 接口）的概念。

接口 MemoryHeapBase 在 Server 端的实现过程中，可以将所有涉及的类分为如下 3 种类型。

☑ 业务相关类：即和匿名共享内存操作相关的类，包括 MemoryHeapBase、BnMemoryHeap、IMemoryHeap。
☑ Binder 进程通信类：即和 Binder 进程通信机制相关的类，包括 IInterface、BnInterface、IBinder、BBinder、ProcessState 和 IPCThreadState。
☑ 智能指针类：RefBase。

在上述 3 种类型中，Binder 进程通信类和智能指针类将在本书后面的章节中进行讲解。在接口 IMemoryBase 中定义了和操作匿名共享内存的几个方法，此接口在文件 frameworks/native/include/binder/IMemory.h 中定义，定义代码如下所示。

```
class IMemoryHeap : public IInterface
{
public:
    DECLARE_META_INTERFACE(MemoryHeap);

    // flags returned by getFlags()
    enum {
        READ_ONLY     = 0x00000001
    };

    virtual int             getHeapID() const = 0;
    virtual void*           getBase() const = 0;
    virtual size_t          getSize() const = 0;
    virtual uint32_t        getFlags() const = 0;
    virtual uint32_t        getOffset() const = 0;

    // these are there just for backward source compatibility
    int32_t heapID() const { return getHeapID(); }
    void*     base() const    { return getBase(); }
    size_t    virtualSize() const { return getSize(); }
};
```

在上述定义代码中有如下 3 个重要的成员函数。

☑ getHeapID：功能是获得匿名共享内存块的打开文件描述符。
☑ getBase：功能是获得匿名共享内存块的基地址，通过这个地址可以在程序中直接访问这块共享内存。
☑ getSize：功能是获得匿名共享内存块的大小。

类 BnMemoryHeap 是一个本地对象类，当 Client 端引用请求 Server 端对象执行命令时，Binder 系统就会调用类 BnMemoryHeap 的成员函数 onTransact()执行具体的命令。函数 onTransact()在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
status_t BnMemory::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case GET_MEMORY: {
            CHECK_INTERFACE(IMemory, data, reply);
            ssize_t offset;
            size_t size;
            reply->writeStrongBinder( getMemory(&offset, &size)->asBinder() );
            reply->writeInt32(offset);
            reply->writeInt32(size);
            return NO_ERROR;
```

```
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
```

类 MemoryHeapBase 继承了类 BnMemoryHeap，作为 Binder 机制中的 Server 角色需要实现 IMemoryBase 接口，主要功能是实现类 IMemoryBase 中列出的成员函数，描述了一块匿名共享内存服务。类在文件 frameworks/native/include/binder/MemoryHeapBase.h 中定义，具体实现代码如下所示。

```
class MemoryHeapBase : public virtual BnMemoryHeap
{
public:
    enum {
        READ_ONLY = IMemoryHeap::READ_ONLY,
        // memory won't be mapped locally, but will be mapped in the remote
        // process
        DONT_MAP_LOCALLY = 0x00000100,
        NO_CACHING = 0x00000200
    };

    /*
     * maps the memory referenced by fd. but DOESN'T take ownership
     * of the filedescriptor (it makes a copy with dup())
     */
    MemoryHeapBase(int fd, size_t size, uint32_t flags = 0, uint32_t offset = 0);

    /*
     * maps memory from the given device
     */
    MemoryHeapBase(const char* device, size_t size = 0, uint32_t flags = 0);

    /*
     * maps memory from ashmem, with the given name for debugging
     */
    MemoryHeapBase(size_t size, uint32_t flags = 0, char const* name = NULL);

    virtual ~MemoryHeapBase();

    /* implement IMemoryHeap interface */
    virtual int             getHeapID() const;

    /* virtual address of the heap. returns MAP_FAILED in case of error */
    virtual void*           getBase() const;

    virtual size_t          getSize() const;
    virtual uint32_t        getFlags() const;
    virtual uint32_t        getOffset() const;

    const char*             getDevice() const;

    /* this closes this heap -- use carefully */
```

```
    void dispose();

    /* this is only needed as a workaround, use only if you know
     * what you are doing */
    status_t setDevice(const char* device) {
        if (mDevice == 0)
            mDevice = device;
        return mDevice ? NO_ERROR : ALREADY_EXISTS;
    }

protected:
            MemoryHeapBase();
    // init() takes ownership of fd
    status_t init(int fd, void *base, int size,
            int flags = 0, const char* device = NULL);

private:
    status_t mapfd(int fd, size_t size, uint32_t offset = 0);

    int         mFD;    //是一个文件描述符，是在打开设备文件/dev/ashmem 后得到的，能够描述一个匿名共
享内存块
    size_t      mSize;  //内存块的大小
    void*       mBase;  //内存块的映射地址
    uint32_t    mFlags; //内存块的访问保护位
    const char* mDevice;
    bool        mNeedUnmap;
    uint32_t    mOffset;
};
```

类 MemoryHeapBase 在文件 frameworks/native/libs/binder/MemoryHeapBase.cpp 中实现，其核心功能是包含了一块匿名共享内存，对应代码如下所示。

```
MemoryHeapBase::MemoryHeapBase(size_t size, uint32_t flags, char const * name)
    : mFD(-1), mSize(0), mBase(MAP_FAILED), mFlags(flags),
      mDevice(0), mNeedUnmap(false), mOffset(0)
{
    const size_t pagesize = getpagesize();
    size = ((size + pagesize-1) & ~(pagesize-1));
    int fd = ashmem_create_region(name == NULL ? "MemoryHeapBase" : name, size);
    ALOGE_IF(fd<0, "error creating ashmem region: %s", strerror(errno));
    if (fd >= 0) {
        if (mapfd(fd, size) == NO_ERROR) {
            if (flags & READ_ONLY) {
                ashmem_set_prot_region(fd, PROT_READ);
            }
        }
    }
}
```

参数说明如下。

- ☑ size：表示要创建的匿名共享内存的大小。
- ☑ flags：设置这块匿名共享内存的属性，例如可读写、只读等。
- ☑ name：此参数只是作为调试信息使用的，用于标识匿名共享内存的名字，可以是空值。

接下来看 MemoryHeapBase 的成员函数 mapfd()，其功能是将得到的匿名共享内存的文件描述符映射到进程地址空间。函数 mapfd()在文件 frameworks/native/libs/binder/MemoryHeapBase.cpp 中定义，具体实现代码如下所示。

```
status_t MemoryHeapBase::mapfd(int fd, size_t size, uint32_t offset)
{
    if (size == 0) {
        // try to figure out the size automatically
#ifdef HAVE_ANDROID_OS
        // first try the PMEM ioctl
        pmem_region reg;
        int err = ioctl(fd, PMEM_GET_TOTAL_SIZE, &reg);
        if (err == 0)
            size = reg.len;
#endif
        if (size == 0) { // try fstat
            struct stat sb;
            if (fstat(fd, &sb) == 0)
                size = sb.st_size;
        }
        // if it didn't work, let mmap() fail.
    }

    if ((mFlags & DONT_MAP_LOCALLY) == 0) {//条件为 true 时执行系统调用 mmap 来执行内存映射的操作
        void* base = (uint8_t*)mmap(
0,//表示由内核来决定这个匿名共享内存文件在进程地址空间的起始位置
 size,//表示要映射的匿名共享内文件的大小
PROT_READ|PROT_WRITE,//表示这个匿名共享内存是可读写的
 MAP_SHARED,
 fd, //指定要映射的匿名共享内存的文件描述符
 offset//表示要从这个文件的哪个偏移位置开始映射
);
        if (base == MAP_FAILED) {
            ALOGE("mmap(fd=%d, size=%u) failed (%s)",
                    fd, uint32_t(size), strerror(errno));
            close(fd);
            return -errno;
        }
        //ALOGD("mmap(fd=%d, base=%p, size=%lu)", fd, base, size);
        mBase = base;
        mNeedUnmap = true;
    } else  {
        mBase = 0; // not MAP_FAILED
        mNeedUnmap = false;
    }
    mFD = fd;
    mSize = size;
    mOffset = offset;
    return NO_ERROR;
}
```

这样在调用函数 mapfd()后，会进入内核空间的 Ashmem 驱动程序模块中执行函数 ashmem_map()。有

关函数 ashmem_map()的具体实现过程，已在 7.1 节的内容中进行了详细讲解。

最后看成员函数 getHeapID()、getBase()和 getSize()的具体实现，具体实现代码如下所示。

```
int MemoryHeapBase::getHeapID() const {
    return mFD;
}
void* MemoryHeapBase::getBase() const {
    return mBase;
}
size_t MemoryHeapBase::getSize() const {
    return mSize;
}
```

7.2.2 接口 MemoryHeapBase 的客户端实现

在接口 MemoryHeapBase 的客户端的实现过程中，可以将所有涉及的类分为如下 3 种类型。

- ☑ 业务相关类：即和匿名共享内存操作相关的类，包括 BpMemoryHeap、IMemoryHeap。
- ☑ Binder 进程通信类：即和 Binder 进程通信机制相关的类，包括 IInterface、BpInterface、IBinder、BpBinder、ProcessState、BpRefBase 和 IPCThreadState。
- ☑ 智能指针类：RefBase。

在上述 3 种类型中，Binder 进程通信类和智能指针类将在本书后面的章节中进行讲解，在本章将重点介绍业务相关类。

类 BpMemoryHeap 是类 MemoryHeapBase 在 Client 端进程的远程接口类，当 Client 端进程从 Service Manager 获得了一个 MemoryHeapBase 对象的引用后，会在本地创建一个 BpMemoryHeap 对象来表示这个引用。类 BpMemoryHeap 是从 RefBase 类继承下来的，也要实现 IMemoryHeap 接口，可以和智能指针结合使用。

类 BpMemoryHeap 在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
class BpMemoryHeap : public BpInterface<IMemoryHeap>
{
public:
    BpMemoryHeap(const sp<IBinder>& impl);
    virtual ~BpMemoryHeap();

    virtual int getHeapID() const;
    virtual void* getBase() const;
    virtual size_t getSize() const;
    virtual uint32_t getFlags() const;
    virtual uint32_t getOffset() const;

private:
    friend class IMemory;
    friend class HeapCache;

    // for debugging in this module
    static inline sp<IMemoryHeap> find_heap(const sp<IBinder>& binder) {
        return gHeapCache->find_heap(binder);
    }
    static inline void free_heap(const sp<IBinder>& binder) {
        gHeapCache->free_heap(binder);
```

```
    }
    static inline sp<IMemoryHeap> get_heap(const sp<IBinder>& binder) {
        return gHeapCache->get_heap(binder);
    }
    static inline void dump_heaps() {
        gHeapCache->dump_heaps();
    }

    void assertMapped() const;
    void assertReallyMapped() const;

    mutable volatile int32_t mHeapId;
    mutable void*           mBase;
    mutable size_t          mSize;
    mutable uint32_t        mFlags;
    mutable uint32_t        mOffset;
    mutable bool            mRealHeap;
    mutable Mutex           mLock;
};
```

类 BpMemoryHeap 对应的构造函数是 BpMemoryHeap()，具体实现代码如下所示。

```
BpMemoryHeap::BpMemoryHeap(const sp<IBinder>& impl)
    : BpInterface<IMemoryHeap>(impl),
        mHeapId(-1), mBase(MAP_FAILED), mSize(0), mFlags(0), mRealHeap(false)
{
}
```

成员函数 getHeapID()、getBase()和 getSize()的实现代码如下所示。

```
int BpMemoryHeap::getHeapID() const {
    assertMapped();
    return mHeapId;
}

void* BpMemoryHeap::getBase() const {
    assertMapped();
    return mBase;
}

size_t BpMemoryHeap::getSize() const {
    assertMapped();
    return mSize;
}
```

在使用上述成员函数之前，通过调用函数 assertMapped()来确保在 Client 端已经准备好了匿名共享内存。函数 assertMapped()在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
void BpMemoryHeap::assertMapped() const
{
    if (mHeapId == -1) {
        sp<IBinder> binder(const_cast<BpMemoryHeap*>(this)->asBinder());
        sp<BpMemoryHeap> heap(static_cast<BpMemoryHeap*>(find_heap(binder).get()));
        heap->assertReallyMapped();
        if (heap->mBase != MAP_FAILED) {
            Mutex::Autolock _l(mLock);
```

```
            if (mHeapId == -1) {
                mBase = heap->mBase;
                mSize = heap->mSize;
                android_atomic_write( dup( heap->mHeapId ), &mHeapId );
            }
        } else {
            // something went wrong
            free_heap(binder);
        }
    }
}
```

类 HeapCache 在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
class HeapCache : public IBinder::DeathRecipient
{
public:
    HeapCache();
    virtual ~HeapCache();

    virtual void binderDied(const wp<IBinder>& who);

    sp<IMemoryHeap> find_heap(const sp<IBinder>& binder);
    void free_heap(const sp<IBinder>& binder);
    sp<IMemoryHeap> get_heap(const sp<IBinder>& binder);
    void dump_heaps();

private:
    // For IMemory.cpp
    struct heap_info_t {
        sp<IMemoryHeap> heap;
        int32_t         count;
    };

    void free_heap(const wp<IBinder>& binder);

    Mutex mHeapCacheLock;
    KeyedVector< wp<IBinder>, heap_info_t > mHeapCache;
};
```

在上述代码中定义了成员变量 mHeapCache，功能是维护进程内的所有 BpMemoryHeap 对象。另外还提供了函数 find_heap()和函数 get_heap()来查找内部所维护的 BpMemoryHeap 对象，这两个函数的具体说明如下。

- ☑ 函数 find_heap()：如果在 mHeapCache 找不到相应的 BpMemoryHeap 对象，则把 BpMemoryHeap 对象加入到 mHeapCache 中。
- ☑ 函数 get_heap()：不会自动把 BpMemoryHeap 对象加入到 mHeapCache 中。

接下来看函数 find_heap()，首先以传进来的参数 binder 作为关键字在 mHeapCache 中查找，查找是否存在对应的 heap_info 对象 info。

- ☑ 如果有，增加引用计数 info.count 的值，表示此 BpBinder 对象多了一个使用者。
- ☑ 如果没有，创建一个放到 mHeapCache 中的 heap_info 对象 info。

函数 find_heap()在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
sp<IMemoryHeap> HeapCache::find_heap(const sp<IBinder>& binder)
```

```
{
    Mutex::Autolock _l(mHeapCacheLock);
    ssize_t i = mHeapCache.indexOfKey(binder);
    if (i>=0) {
        heap_info_t& info = mHeapCache.editValueAt(i);
        ALOGD_IF(VERBOSE,
                "found binder=%p, heap=%p, size=%d, fd=%d, count=%d",
                binder.get(), info.heap.get(),
                static_cast<BpMemoryHeap*>(info.heap.get())->mSize,
                static_cast<BpMemoryHeap*>(info.heap.get())->mHeapId,
                info.count);
        android_atomic_inc(&info.count);
        return info.heap;
    } else {
        heap_info_t info;
        info.heap = interface_cast<IMemoryHeap>(binder);
        info.count = 1;
        //ALOGD("adding binder=%p, heap=%p, count=%d",
        //binder.get(), info.heap.get(), info.count);
        mHeapCache.add(binder, info);
        return info.heap;
    }
}
```

由上述实现代码可知，函数 find_heap()是 BpMemoryHeap 的成员函数，能够调用全局变量 gHeapCache 执行查找的操作，对应的实现代码如下所示。

```
class BpMemoryHeap : public BpInterface<IMemoryHeap>
{
...
private:
    static inline sp<IMemoryHeap> find_heap(const sp<IBinder>& binder) {
        return gHeapCache->find_heap(binder);
    }
```

通过调用函数 find_heap()得到 BpMemoryHeap 对象中的函数 assertReallyMapped()，这样可以确认它内部的匿名共享内存是否已经映射到进程空间。函数 assertReallyMapped()在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
void BpMemoryHeap::assertReallyMapped() const
{
    if (mHeapId == -1) {

        // remote call without mLock held, worse case scenario, we end up
        // calling transact() from multiple threads, but that's not a problem,
        // only mmap below must be in the critical section.

        Parcel data, reply;
        data.writeInterfaceToken(IMemoryHeap::getInterfaceDescriptor());
        status_t err = remote()->transact(HEAP_ID, data, &reply);
        int parcel_fd = reply.readFileDescriptor();
        ssize_t size = reply.readInt32();
        uint32_t flags = reply.readInt32();
        uint32_t offset = reply.readInt32();
```

```
        ALOGE_IF(err, "binder=%p transaction failed fd=%d, size=%ld, err=%d (%s)",
                asBinder().get(), parcel_fd, size, err, strerror(-err));

        int fd = dup( parcel_fd );
        ALOGE_IF(fd==-1, "cannot dup fd=%d, size=%ld, err=%d (%s)",
                parcel_fd, size, err, strerror(errno));

        int access = PROT_READ;
        if (!(flags & READ_ONLY)) {
            access |= PROT_WRITE;
        }

        Mutex::Autolock _l(mLock);
        if (mHeapId == -1) {
            mRealHeap = true;
            mBase = mmap(0, size, access, MAP_SHARED, fd, offset);
            if (mBase == MAP_FAILED) {
                ALOGE("cannot map BpMemoryHeap (binder=%p), size=%ld, fd=%d (%s)",
                        asBinder().get(), size, fd, strerror(errno));
                close(fd);
            } else {
                mSize = size;
                mFlags = flags;
                mOffset = offset;
                android_atomic_write(fd, &mHeapId);
            }
        }
    }
}
```

7.2.3 接口 MemoryBase 的服务器端实现

接口 MemoryBase 是建立在接口 MemoryHeapBase 的基础上的，两者都可以作为一个 Binder 对象在进程之间实现数据共享。首先分析类 MemoryBase 在 Server 端的实现，MemoryBase 在 Server 端只是简单地封装了 MemoryHeapBase 的实现。类 MemoryBase 在 Server 端的实现和类 MemoryHeapBase 在 Server 端的实现类似，只需在整个类图结构中实现如下转换即可。

- ☑ 把类 IMemory 换成类 IMemoryHeap。
- ☑ 把类 BnMemory 换成类 BnMemoryHeap。
- ☑ 把类 MemoryBase 换成类 MemoryHeapBase。

类 IMemory 在文件 frameworks/native/include/binder/IMemory.h 中实现，功能是定义类 MemoryBase 所需要的实现接口。类 IMemory 的实现代码如下所示。

```
class IMemory : public IInterface
{
public:
    DECLARE_META_INTERFACE(Memory);
    virtual sp<IMemoryHeap> getMemory(ssize_t* offset=0, size_t* size=0) const = 0;
    // helpers
    void* fastPointer(const sp<IBinder>& heap, ssize_t offset) const;
```

```
    void* pointer() const;
    size_t size() const;
    ssize_t offset() const;
};
```

在类 IMemory 中定义了如下成员函数。

☑ getMemory：功能是获取内部的 MemoryHeapBase 对象的 IMemoryHeap 接口。

☑ pointer()：功能是获取内部所维护的匿名共享内存的基地址。

☑ size()：功能是获取内部所维护的匿名共享内存的大小。

☑ offset()：功能是获取内部所维护的匿名共享内存在整个匿名共享内存中的偏移量。

类 IMemory 在本身定义过程中实现了 3 个成员函数 pointer()、size()和 offset()，其子类 MemoryBase 只需实现成员函数 getMemory()即可。类 IMemory 的具体实现在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
void* IMemory::pointer() const {
    ssize_t offset;
    sp<IMemoryHeap> heap = getMemory(&offset);
    void* const base = heap!=0 ? heap->base() : MAP_FAILED;
    if (base == MAP_FAILED)
        return 0;
    return static_cast<char*>(base) + offset;
}

size_t IMemory::size() const {
    size_t size;
    getMemory(NULL, &size);
    return size;
}
ssize_t IMemory::offset() const {
    ssize_t offset;
    getMemory(&offset);
    return offset;
}
```

类 MemoryBase 是一个本地 Binder 对象类，在文件 frameworks/native/include/binder/MemoryBase.h 中声明，具体定义代码如下所示。

```
class MemoryBase : public BnMemory
{
public:
    MemoryBase(const sp<IMemoryHeap>& heap, ssize_t offset, size_t size);
    virtual ~MemoryBase();
    virtual sp<IMemoryHeap> getMemory(ssize_t* offset, size_t* size) const;
protected:
    size_t getSize() const { return mSize; }
    ssize_t getOffset() const { return mOffset; }
    const sp<IMemoryHeap>& getHeap() const { return mHeap; }
private:
    size_t          mSize;
    ssize_t         mOffset;
    sp<IMemoryHeap> mHeap;
};
}; // namespace android
#endif // ANDROID_MEMORY_BASE_H
```

类 MemoryBase 的具体实现在文件 frameworks/native/libs/binder/MemoryBase.cpp 中定义，具体实现代码如下所示。

```
MemoryBase::MemoryBase(const
 sp<IMemoryHeap>& heap,//指向的 MemoryHeapBase 对象，真正的匿名共享内存就是由它来维护的
 ssize_t offset,//表示这个 MemoryBase 对象所要维护的这部分匿名共享内存在整个匿名共享内存块中的起始位置
 size_t size//表示这个 MemoryBase 对象所要维护的这部分匿名共享内存的大小
)
    : mSize(size), mOffset(offset), mHeap(heap)
{
}
//功能是返回内部的 MemoryHeapBase 对象的 IMemoryHeap 接口。如果传进来的参数 offset 和 size 不为 NULL，会把其内部维护的这部分匿名共享内存，在整个匿名共享内存块中的偏移位置以及这部分匿名共享内存的大小返回给调用者
sp<IMemoryHeap> MemoryBase::getMemory(ssize_t* offset, size_t* size) const
{
    if (offset) *offset = mOffset;
    if (size)    *size = mSize;
    return mHeap;
}
```

7.2.4 接口 MemoryBase 的客户端实现

类 MemoryBase 在 Client 端的实现与类 MemoryHeapBase 在 Client 端的实现类似，只需要进行如下所示的类转换即可成为 MemoryHeapBase 在 Client 端的实现。

☑ 把类 IMemory 换成类 IMemoryHeap。

☑ 把类 BpMemory 换成类 BpMemoryHeap。

类 BpMemory 用于描述类 MemoryBase 服务的代理对象，在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
class BpMemory : public BpInterface<IMemory>
{
public:
    BpMemory(const sp<IBinder>& impl);
    virtual ~BpMemory();
    virtual sp<IMemoryHeap> getMemory(ssize_t* offset=0, size_t* size=0) const;

private:
    mutable sp<IMemoryHeap> mHeap;//类型为 IMemoryHeap，它指向的是一个 BpMemoryHeap 对象
    mutable ssize_t mOffset;//表示 BpMemory 对象所要维护的匿名共享内存在整个匿名共享内存块中的起始位置
    mutable size_t mSize;//表示这个 BpMemory 对象所要维护的这部分匿名共享内存的大小
};
```

类 BpMemory 中的成员函数 getMemory()在文件 frameworks/native/libs/binder/IMemory.cpp 中定义，具体实现代码如下所示。

```
sp<IMemoryHeap> BpMemory::getMemory(ssize_t* offset, size_t* size) const
{
    if (mHeap == 0) {
        Parcel data, reply;
        data.writeInterfaceToken(IMemory::getInterfaceDescriptor());
        if (remote()->transact(GET_MEMORY, data, &reply) == NO_ERROR) {
```

```
                sp<IBinder> heap = reply.readStrongBinder();
                ssize_t o = reply.readInt32();
                size_t s = reply.readInt32();
                if (heap != 0) {
                    mHeap = interface_cast<IMemoryHeap>(heap);
                    if (mHeap != 0) {
                        mOffset = o;
                        mSize = s;
                    }
                }
            }
        }
        if (offset) *offset = mOffset;
        if (size) *size = mSize;
        return mHeap;
}
```

如果成员变量 mHeap 的值为 NULL，表示此 BpMemory 对象还没有建立好匿名共享内存，此时会调用一个 Binder 进程去 Server 端请求匿名共享内存信息。通过引用信息中的 Server 端的 MemoryHeapBase 对象的引用 heap，可以在 Client 端进程中创建一个 BpMemoryHeap 远程接口，最后将这个 BpMemoryHeap 远程接口保存在成员变量 mHeap 中，同时从 Server 端获得的信息还包括这块匿名共享内存在整个匿名共享内存中的偏移位置及大小。

7.3　实现 Java 访问的接口层

分析完匿名共享内存的 C++访问接口层后，本节开始分析其 Java 访问接口层的实现过程。在 Android 应用程序框架层中，通过使用接口 MemoryFile 来封装匿名共享内存文件的创建和使用。接口 MemoryFile 在文件 frameworks/base/core/java/android/os/MemoryFile.java 中定义，具体实现代码如下所示。

```
public class MemoryFile
{
    private static String TAG = "MemoryFile";

    // mmap(2) protection flags from <sys/mman.h>
    private static final int PROT_READ = 0x1;
    private static final int PROT_WRITE = 0x2;

    private static native FileDescriptor native_open(String name, int length) throws IOException;
    // returns memory address for ashmem region
    private static native int native_mmap(FileDescriptor fd, int length, int mode)
            throws IOException;
    private static native void native_munmap(int addr, int length) throws IOException;
    private static native void native_close(FileDescriptor fd);
    private static native int native_read(FileDescriptor fd, int address, byte[] buffer,
            int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
    private static native void native_write(FileDescriptor fd, int address, byte[] buffer,
            int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
    private static native void native_pin(FileDescriptor fd, boolean pin) throws IOException;
```

```
    private static native int native_get_size(FileDescriptor fd) throws IOException;

    private FileDescriptor mFD;           // ashmem file descriptor
    private int mAddress;     // address of ashmem memory
    private int mLength;      // total length of our ashmem region
    private boolean mAllowPurging = false;    // true if our ashmem region is unpinned

    /**
     * Allocates a new ashmem region. The region is initially not purgable.
     *
     * @param name optional name for the file (can be null).
     * @param length of the memory file in bytes.
     * @throws IOException if the memory file could not be created.
     */
    public MemoryFile(String name, int length) throws IOException {
        mLength = length;
        mFD = native_open(name, length);
        if (length > 0) {
            mAddress = native_mmap(mFD, length, PROT_READ | PROT_WRITE);
        } else {
            mAddress = 0;
        }
    }
```

在上述代码中，构造方法 MemoryFile()以指定的字符串调用了 JNI 方法 native_open()，目的是建立一个匿名共享内存文件，这样可以得到一个文件描述符。然后使用这个文件描述符为参数调用 JNI 方法 native_mmap()，并把匿名共享内存文件映射到进程空间中，这样就可以通过映射得到地址空间的方式直接访问内存数据。

JNI 函数 android_os_MemoryFile_get_size()在文件 frameworks/base/core/jni/android_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jint android_os_MemoryFile_get_size(JNIEnv* env, jobject clazz,
        jobject fileDescriptor) {
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    // Use ASHMEM_GET_SIZE to find out if the fd refers to an ashmem region.
    // ASHMEM_GET_SIZE should succeed for all ashmem regions, and the kernel
    // should return ENOTTY for all other valid file descriptors
    int result = ashmem_get_size_region(fd);
    if (result < 0) {
        if (errno == ENOTTY) {
            // ENOTTY means that the ioctl does not apply to this object,
            // i.e., it is not an ashmem region.
            return (jint) -1;
        }
        // Some other error, throw exception
        jniThrowIOException(env, errno);
        return (jint) -1;
    }
    return (jint) result;
}
```

JNI 函数 android_os_MemoryFile_open()在文件 frameworks/base/core/jni/android_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jobject android_os_MemoryFile_open(JNIEnv* env, jobject clazz, jstring name, jint length)
{
    const char* namestr = (name ? env->GetStringUTFChars(name, NULL) : NULL);

    int result = ashmem_create_region(namestr, length);

    if (name)
        env->ReleaseStringUTFChars(name, namestr);

    if (result < 0) {
        jniThrowException(env, "java/io/IOException", "ashmem_create_region failed");
        return NULL;
    }

    return jniCreateFileDescriptor(env, result);
}
```

JNI 函数 android_os_MemoryFile_mmap()在文件 frameworks/base/core/jni/android_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jint android_os_MemoryFile_mmap(JNIEnv* env, jobject clazz, jobject fileDescriptor,
        jint length, jint prot)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    jint result = (jint)mmap(NULL, length, prot, MAP_SHARED, fd, 0);
    if (!result)
        jniThrowException(env, "java/io/IOException", "mmap failed");
    return result;
}
```

在文件 frameworks/base/core/java/android/os/MemoryFile.java 中，类 MemoryFile 的成员函数 readBytes 的功能是读取某一块匿名共享内存的内容，具体实现代码如下所示。

```
public int readBytes(byte[] buffer, int srcOffset, int destOffset, int count)
        throws IOException {
    if (isDeactivated()) {
        throw new IOException("Can't read from deactivated memory file.");
    }
    if (destOffset < 0 || destOffset > buffer.length || count < 0
            || count > buffer.length - destOffset
            || srcOffset < 0 || srcOffset > mLength
            || count > mLength - srcOffset) {
        throw new IndexOutOfBoundsException();
    }
    return native_read(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
```

在文件 frameworks/base/core/java/android/os/MemoryFile.java 中，类 MemoryFile 的成员函数 writeBytes()的功能是写入某一块匿名共享内存的内容，具体实现代码如下所示。

```
public void writeBytes(byte[] buffer, int srcOffset, int destOffset, int count)
            throws IOException {
```

```
        if (isDeactivated()) {
            throw new IOException("Can't write to deactivated memory file.");
        }
        if (srcOffset < 0 || srcOffset > buffer.length || count < 0
                || count > buffer.length - srcOffset
                || destOffset < 0 || destOffset > mLength
                || count > mLength - destOffset) {
            throw new IndexOutOfBoundsException();
        }
        native_write(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
```

在文件 frameworks/base/core/java/android/os/MemoryFile.java 中，类 MemoryFile 的成员函数 isDeactivated()的功能是保证匿名共享内存已经被映射到进程的地址空间中，具体实现代码如下所示。

```
void deactivate() {
        if (!isDeactivated()) {
            try {
                native_munmap(mAddress, mLength);
                mAddress = 0;
            } catch (IOException ex) {
                Log.e(TAG, ex.toString());
            }
}
}
private boolean isDeactivated() {
        return mAddress == 0;
}
```

JNI 函数 native_read()和 native_write()分别由位于 C++层的函数 android_os_MemoryFile_read()和 android_os_MemoryFile_write()实现，这两个 C++的函数在文件 frameworks/base/core/jni/android_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jint android_os_MemoryFile_read(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }
    env->SetByteArrayRegion(buffer, destOffset, count, (const jbyte *)address + srcOffset);
    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}
static jint android_os_MemoryFile_write(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
```

```
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }
    env->GetByteArrayRegion(buffer, srcOffset, count, (jbyte *)address + destOffset);
    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}
```

7.4　实战演练——读取内核空间的数据

本实例的功能是在驱动程序中将 Buffer 指向的内存空间映射到用户空间中，然后在去掉程序中向 Buffer 中写入一个字符串，最后在用户程序中读取这个字符串的数据。

（1）编写驱动程序文件 mmap_shared.c，其中结构体指针 vm_operations_struct 是在设备文件 mmap 函数 demo_mmap()中指定的，当应用程序调用 mmap()函数申请内存映射时会调用这个函数。因为在调用函数 demo_mmap()时没有指定结构体指针 vm_operations_struct，所以不会调用函数 vm_operations_struct.open()，而在应用程序调用 munmap()函数时调用函数 vm_operations_struct.close()。文件 mmap_shared.c 的具体实现代码如下所示。

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
//定义设备文件名
#define DEVICE_NAME "mmap_shared"
#define BUFFER_SIZE 4096
static char *buffer;
static void demo_vma_open(struct vm_area_struct *vma)
{
     printk(KERN_INFO"VMA open.\n");
}
static void demo_vma_close(struct vm_area_struct *vma)
{
     printk(KERN_INFO"VMA close.\n");
}
static struct vm_operations_struct remap_vm_ops =
{ .open = demo_vma_open, .close = demo_vma_close};
static int demo_mmap(struct file *filp, struct vm_area_struct *vma)
{
     unsigned long physics = virt_to_phys((void*) (unsigned long) buffer); //((unsigned long )buffer)-PAGE_
```

```
OFFSET;
        unsigned long mypfn = physics >> PAGE_SHIFT;
        unsigned long vmsize = vma->vm_end - vma->vm_start;
        printk(KERN_INFO"demo_mmap called\n");
        if (vmsize > BUFFER_SIZE)
                return -EINVAL;
        vma->vm_ops = &remap_vm_ops;
        vma->vm_flags |= VM_RESERVED;
        demo_vma_open(vma);
        if (remap_pfn_range(vma, vma->vm_start, mypfn, vmsize, vma->vm_page_prot))
                return -EAGAIN;
        return 0;
    }
    static struct file_operations dev_fops =
    { .owner = THIS_MODULE, .mmap = demo_mmap };
    //描述设备文件的信息
    static struct miscdevice misc =
    { .minor = MISC_DYNAMIC_MINOR, .name = DEVICE_NAME, .fops = &dev_fops };
    static int __init demo_init(void)
    {
        int ret;
        struct page *page;
        //建立设备文件
        ret = misc_register(&misc);
        buffer = kmalloc(BUFFER_SIZE, GFP_KERNEL);
        for (page = virt_to_page(buffer); page < virt_to_page(buffer + BUFFER_SIZE);
                      page++)
        {
                //将当前页设为保留状态
                SetPageReserved(page);
        }memset(buffer, 0, BUFFER_SIZE);
        strcpy(buffer, "mmap_shared_success!\n");

        printk(KERN_INFO "demo_init.\n");
        return ret;
    }

    static void __exit demo_exit(void)
    {
        struct page *page;
        //删除设备文件
        misc_deregister(&misc);
        for (page = virt_to_page(buffer); page < virt_to_page(buffer + BUFFER_SIZE);
                      page++)
        {
                //清除页的保留状态
                ClearPageReserved(page);
        }
        printk(KERN_INFO "demo_exit.\n");
    }
    MODULE_LICENSE("GPL");
```

```
    module_init(demo_init);
    module_exit(demo_exit);
```

通过上述驱动程序，即可在 mmap()函数中实现内存映射的初始化工作，也可以在 mmap()函数中调用 vm_operations_struct.open()。

（2）编写用户应用程序 user_mmap.c，功能是读取内存映射，如果申请内存映射成功，则通过 mmap()函数返回内核空间映射到用户空间内存的首地址。文件 user_mmap.c 的具体实现代码如下所示。

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/mman.h>
#include <stdio.h>
#define PAGE_SIZE (4*1024)
int main()
{
        int fd;
        void *start;
        fd = open("/dev/mmap_shared", O_RDWR);
        start = mmap(NULL, PAGE_SIZE, PROT_READ, MAP_PRIVATE, fd, 0);
        if (start == MAP_FAILED)
        {
                printf("mmap error\n");
                return 0;
        }
        puts(start);
        munmap(start, PAGE_SIZE);
        close(fd);
}
```

此时如果执行 build.sh 脚本文件编译并安装上述文件，然后执行 user_mmap 命令，会在 Linux 终端输出如下所示的信息。

```
mmap_share_succes!
```

第 8 章　搭建测试环境

Android 底层驱动开发和移植工作是一项完整的系统工程，不但需要具备 Linux 内核和驱动开发的知识，而且需要熟悉硬件接口和串联等知识，并且需事先搭建一个测试环境。本书将以当前最流行的 S3C6410 和 Cortex-A8 开发板为例，介绍搭建驱动测试环境的知识，为读者学习本书后面的知识打下基础。

8.1　搭建 S3C6410 开发环境

S3C6410 是基于 Samsung 的 16/32 位 RSIC 微处理器 S3C6410X 的一款开发平台，S3C6410X 是基于 ARM1176JZF-S 核的用于手持、移动等终端设备的通用处理器。因为其价格低廉，并且具备主流的功能，所以是广大初学者学习上手的首要开发板。本节将详细讲解搭建 S3C6410 开发环境的基本知识。

8.1.1　S3C6410 介绍

S3C6410 是一款低功率、高性价比、高性能的用于移动电话和通用处理的 RSIC 处理器。为 2.5G 和 3G 通信服务提供了优化的硬件性能，采用 64/32bit 的内部总线架构，融合了 AXI、AHB、APB 总线。还有很多强大的硬件加速器，包括运动视频处理、音频处理、2D 加速、显示处理和缩放。一个集成的 MFC（Multi-Format video Codec）支持 MPEG4/H.263/H.264 编解码和 VC1 的解码，这个硬件编解码器支持实时的视频会议以及 NRSC 和 PAL 制式的 TV 输出。

S3C6410 内置一个采用最先进技术的 3D 加速器，支持 OpenGL ES 1.1/2.0 和 D3DM API，能实现 4M triangles/s 的 3D 加速。S3C6410 包括优化的外部存储器接口，该接口能满足在高端通信服务中的数据带宽要求。接口分为两路，DRAM 和 Flash/ROM/DRAM 端口。DRAM 端口可以通过配置来支持 Mobile DDR、DDR、Mobile SDRAM、SDRAM。Flash/ROM/DRAM 端口支持 NOR-Flash、NAND-Flash、OneNAND、CF、ROM 等类型的外部存储器任意的 Mobile DDR、DDR、Mobile SDRAM、SDRAM 存储器。

S3C6410 为了降低整个系统的成本和提升总体功能，拥有很多硬件功能外设。

- ☑ Camera 接口。
- ☑ TFT 24bit 真彩色 LCD 控制器。
- ☑ 系统管理单元（电源、时钟等）。
- ☑ 4 通道的 UART。
- ☑ 32 通道的 DMA。
- ☑ 4 通道定时器。
- ☑ 通用 I/O 口。
- ☑ I2S 总线。
- ☑ I2C 总线。
- ☑ USB Host。
- ☑ 高速 USB OTG。
- ☑ SD Host 和高速 MMC 卡接口。

☑ 内部的 PLL 时钟发生器。

8.1.2　OK6410 介绍

对于开发者和学习者来说，都希望寻找一个资料最齐全、功能最稳定、Bug 最少的开发板，在当今市面中，飞凌开发板产品 OK6410 是一个很好的选择，如图 8-1 所示。

飞凌 S3C6410 开发板适用于高端消费类电子产品、工业控制、车载导航、多媒体终端、电子付费终端、行业 PDA、嵌入式教育培训、个人学习等，主要功能如下。



图 8-1　飞凌 OK6410

- ☑ 支持 WinCE、Linux、Android 等系统的一键烧写，宿主机可完美支持 Windows 2000、Windows XP、Windows 7 等主流操作系统。
- ☑ 配件最丰富：支持 WiFi、GPS、GPRS、3G、VGA/TV、摄像头、液晶屏、HDMI 数字高清模块等常用配件，独家支持 CAN、RS485、矩阵键盘、电源管理等选配模块，并提供相应驱动及应用程序。
- ☑ 持续的软件更新：包括 Linux、WinCE、Android 在内的操作系统会不断升级，我们将提供给用户最稳定的软件版本，以及最丰富的应用例程，并开放源码，用户可自由下载。
- ☑ 配套资料是由飞凌工程师精心准备的学习教程和操作手册。首创全图形化引导和视频讲解形式，力图层次清晰、内容丰富、生动易懂。
- ☑ 飞凌为客户提供完善快捷的售后服务，包括开发板技术解答和产品质量保证。

因为本书讲解的是 Android 驱动开发，所以在此只讲解搭建安装 Android 系统的方法。

8.1.3　安装 minicom

minicom 是一款经典的串口工具，能够建立 PC 机和串口设备的通信。OK6410 开发板自带了一个串口，可以通过串口线建立与 PC 的连接。这样通过使用 minicom 工具，即可在 PC 机上查看基于开发板的调试信息。

在 Linux 系统中，安装配置 minicom 的基本流程如下。

（1）通过如下命令测试系统是否支持 USB 转串口，如果直接使用串口线，而没有用到 USB 转串口设备，这一步可以直接跳过。

```
# lsmod | grep usbserial
```

如果有 usbserial，则说明系统支持 USB 转串口。

（2）通过如下命令安装 minicom。

```
apt-get install minicom
apt-get install lrzsz
```

（3）开始配置 minicom，如果用户的系统默认语言不是英文，请执行下面的命令。

```
$LANG=EN
```

这样在接下来的设置中，minicom 将以英文界面呈现，操作起来比较方便。

然后在系统终端输入如下命令进行 minicom 的设置。

```
# minicom -s
```

此时会弹出 Serial port setup 菜单，如图 8-2 所示。

（4）进入 Serial port setup 选项设置串口参数，其界面如图 8-3 所示。



图 8-2　配置主菜单

```
A -    Serial Device      : /dev/ttyS0
B - Lockfile Location     : /var/lock
C -   Callin Program      :
D -  Callout Program      :
E -    Bps/Par/Bits       : 115200 8N1
F - Hardware Flow Control : No
G - Software Flow Control : No

    Change which setting?
```

图 8-3　配置串口设备和传输速率

在上述界面中会发现在每一项前面都有一个英文字母，在键盘上输入字母就会进入相对应的设置项，请读者按照图 8-3 所示进行设置。

注意： 在图 8-3 设置中，因为笔者的 PC 机上有串口，所以可以直接使用串口线，将 Serial Device 设置为/dev/tty/ttyS0。如果没有，则将 Serial Device 选项设置为/dev/tty/USB0。

完成上述设置工作后回车即可返回到 minicom 的主设置界面，如图 8-4 所示。

（5）因为在实际应用中不会频繁地去改动这些参数，所以可以将我们的设置保存为默认参数，方法是选择 Save setup as dfl 选项并回车，如图 8-5 所示。

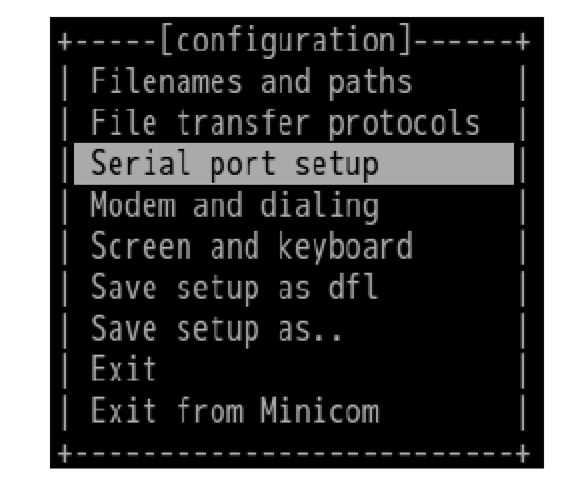


图 8-4　minicom 的主设置界面

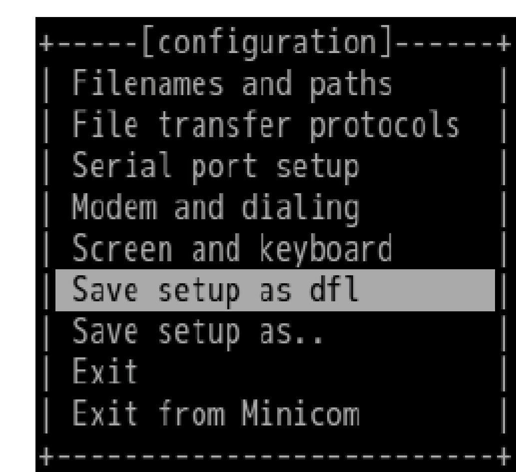


图 8-5　保存设置

（6）选择 Exit 选项并回车，minicom 将开始进行初始化工作，如图 8-6 所示。

当完成初始化操作之后，minicom 会连接到串口，连接到串口的界面如图 8-7 所示。如果此时在串口中有信息输入，将会直接显示出来。

（7）在实际使用中经常需要将 log（日志信息）保存下来，以方便进行 debugging。要想在 minicom 中保存 log，需要按下面的操作步骤进行设置。

首先在 minicom 界面中按 Ctrl+A 快捷键，紧接着再按 Z 键，即可打开 minicom 的帮助界面，如图 8-8 所示。



图 8-6　minicom 初始化

```
Welcome to minicom 2.3

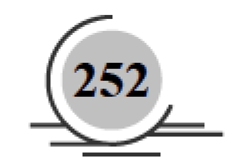
OPTIONS: I18n
Compiled on Sep 25 2009, 23:40:20.
Port /dev/ttyS0

                    Press CTRL-A Z for help on special keys
```

图 8-7　连接到串口的界面



图 8-8　帮助界面



然后打开 Capture on/off 选项，在键盘上输入 L，此时会弹出如图 8-9 所示的界面。



图 8-9　打开 Capture on/off 选项

在默认情况下，会把串口的信息保存在文件 minicom.cap 中，另外读者也可以自己指定保存文件名。如果选择默认将串口信息保存在 minicom.cap 文件中，则在使用 gedit 打开时会发生错误，此时建议使用 gvim 打开。按 Ctrl+A 快捷键再按下 Z 键，进入 minicom 的帮助界面后，输入 X 或者 Q 即可退出 minicom。

注意： X 选项表示退出并重置 minicom，Q 选项则是直接退出不重置 minicom。

8.1.4　烧写 Android 系统

OK6410 开发板的默认系统是 WinCE 6.0，在安装 Android 系统之前需要先卸载 WinCE 6.0 系统，然后再安装 Android 系统。具体实现流程如下所示。

（1）准备 1GB 以上的 SD 卡一张，然后将 SD 卡分为两个区，其中将前一个分区设置为 FAT 格式，将后一个分区设置为 EXT3 格式，并且需要保证 EXT3 分区的大小在 500MB 以上，分区过程在 PC 主机的 Ubuntu 系统下完成。

（2）将 SD 卡插入 PC 机，因为此时 SD 卡会被自动挂载，如图 8-10 所示。

图中左侧矩形框内便是自动挂载 SD 卡，此时需要将 SD 卡确保为卸载状态。单击矩形框区域中的图标，右击并在弹出的快捷菜单中选择 Unmount 命令即可卸载 SD 卡，如图 8-11 所示。

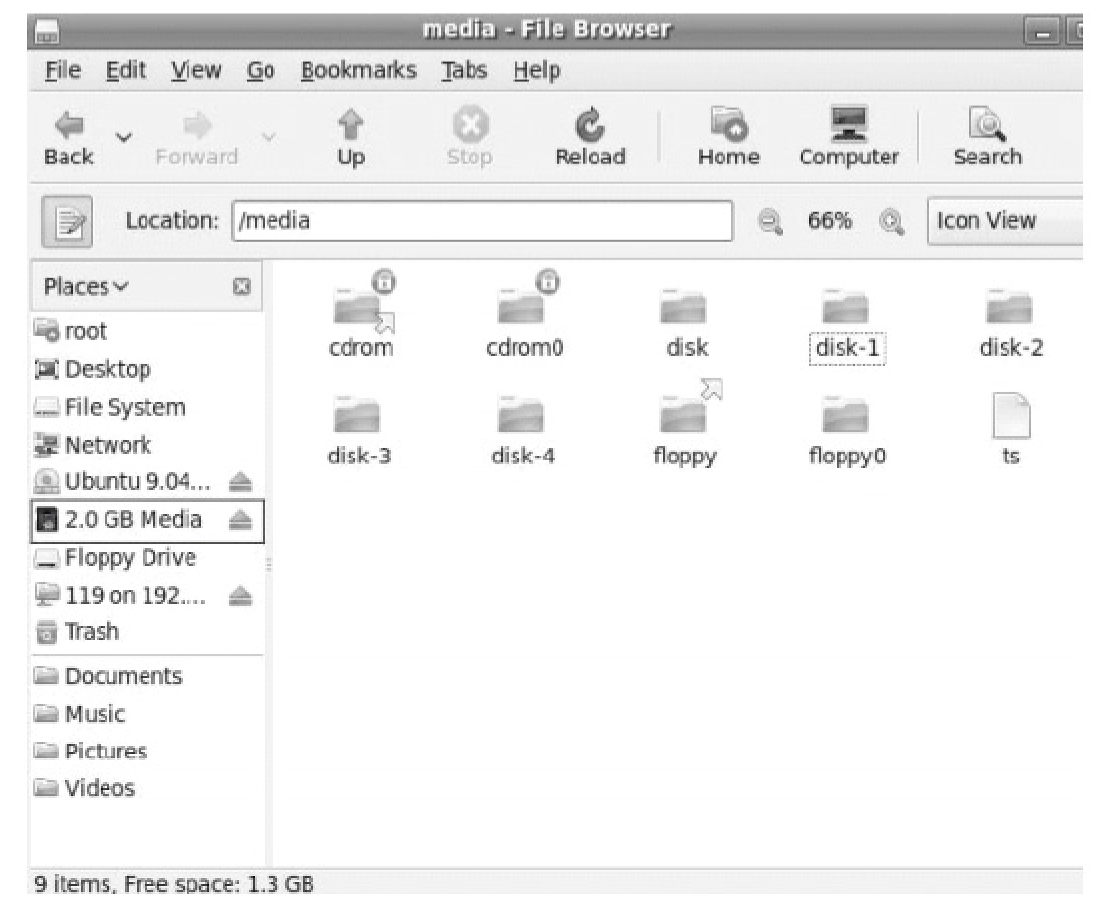




图 8-10　SD 会被自动挂载

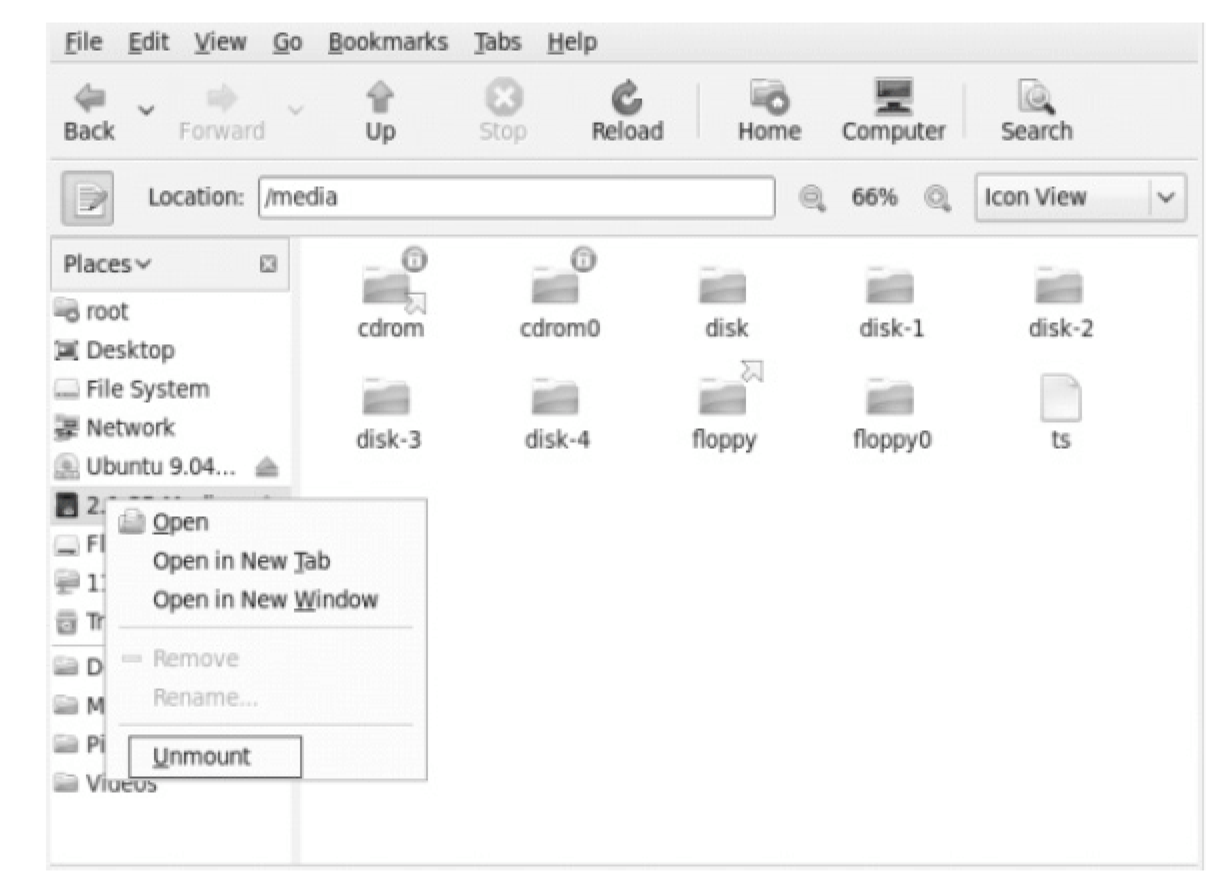


图 8-11　选择 Unmount 命令卸载 SD 卡

卸载后的状态如图 8-12 所示。

（3）打开终端设备，输入如下命令。

```
sudo fdisk /dev/sdb
```

命令截图如图 8-13 所示。

输入 m 后会出现一串选择项，选择 d 删除分区，如图 8-14 所示。

（4）输入如下不同的命令创建第一个分区。

☑　输入 n，回车。

☑　输入 p，回车。

☑ 输入 1，回车。

然后直接回车，输入一个指定大小值，例如 20M，然后按下回车，如图 8-15 所示。

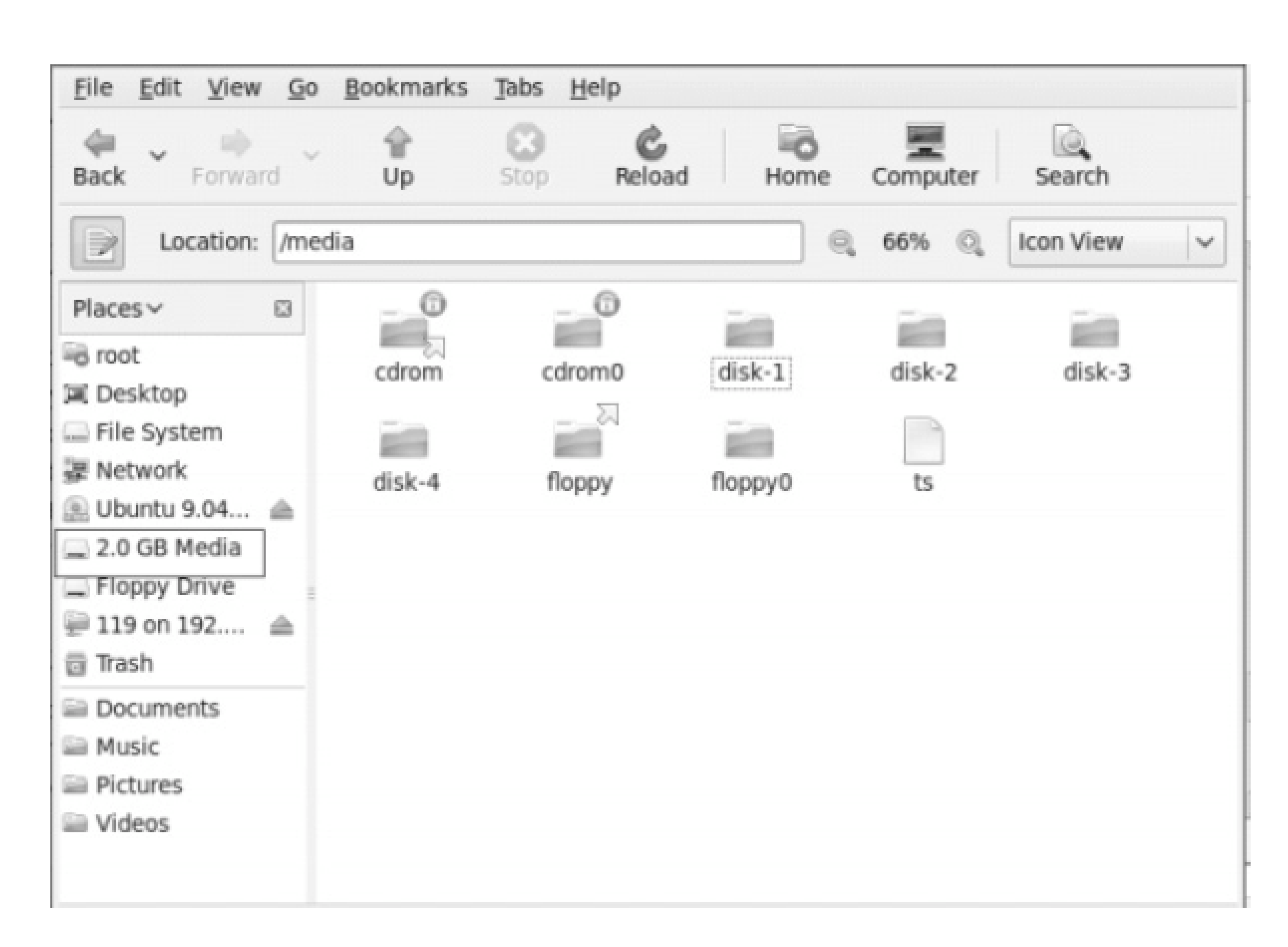


图 8-12　卸载后的状态

```
root@guo-desktop:~#
root@guo-desktop:~# sudo fdisk /dev/sdb

The number of cylinders for this disk is set to 1203.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): m
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)

Command (m for help):
```

图 8-13　输入打开命令

```
Command (m for help): d
Selected partition 1

Command (m for help):
```

图 8-14　删除分区

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1203, default 1):
Using default value 1
Last cylinder, +cylinders or +size{K,M,G} (1-1203, default 1203): 20M

Command (m for help):
```

图 8-15　创建第一个分区

（5）输入如下不同的命令创建第二个分区。

☑ 输入 n，回车。

☑ 输入 p，回车。

☑ 输入 2，回车。

然后直接回车，如图 8-16 所示。

（6）输入如下不同的命令标记第一个分区，如图 8-17 所示。

☑ 输入 a，回车。

☑ 输入 1，回车。

☑ 输入 p，回车。

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 2
First cylinder (21-1203, default 21):
Using default value 21
Last cylinder, +cylinders or +size{K,M,G} (21-1203, default 1203):
Using default value 1203

Command (m for help):
```

图 8-16　创建第二个分区

```
Command (m for help): a
Partition number (1-4): 1

Command (m for help): p

Disk /dev/sdb: 1967 MB, 1967128576 bytes
57 heads, 56 sectors/track, 1203 cylinders
Units = cylinders of 3192 * 512 = 1634304 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *           1          20       31892   83  Linux
/dev/sdb2              21        1203     1888068   83  Linux

Command (m for help):
```

图 8-17　标记第一个分区

（7）输入 w 写入分区表，然后回车，如图 8-18 所示。

（8）开始格式化两个分区，在分区完成后会自动挂载，需要按照前面的步骤先卸载 SD 卡。格式化第一个分区为 vfat 格式，执行命令如下。

```
sudo mkfs.vfat /dev/sdb1
```

命令截图如图 8-19 所示。

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
root@guo-desktop:~#
```

图 8-18 标记第一个分区

```
root@guo-desktop:~# sudo mkfs.vfat /dev/sdb1
mkfs.vfat 3.0.1 (23 Nov 2008)
root@guo-desktop:~#
```

图 8-19 格式化第一个分区为 vfat 格式

（9）格式化第二个分区为 ext3 格式，具体命令如下所示。

```
sudo mkfs.ext3 /dev/sdb2
```

命令截图如图 8-20 所示。

（10）这样就完成了对 SD 卡的分区工作，接下来通过 SD_Fusing_Tool.exe 工具将引导程序 u-boot- sd.bin 和内核 zImage-sd 烧入到 SD 卡中。拔出 SD 卡读卡器，切换到 Windows 环境下，重新插入 SD 卡读卡器。打开 SD_Fusing_Tool.exe 工具，选择烧录文件并设置好读卡器盘符和内核大小，如图 8-21 所示。

在此单击 START 按钮，写入成功后会弹出 Fusing image done 对话框。

（11）开始复制烧写所需的文件，在虚拟机环境下插入 SD 卡读卡器，将 uboot.bin,Image_Nand, android_fs.tar 复制到 ext3 区。如果想正常使用网络功能，需要修改 android_fs.Tar 中的某些配置文件。

```
root@guo-desktop:~#
root@guo-desktop:~# sudo mkfs.ext3 /dev/sdb2
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
118080 inodes, 472017 blocks
23600 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=486539264
15 block groups
32768 blocks per group, 32768 fragments per group
7872 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912

Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 35 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
root@guo-desktop:~#
```

图 8-20 格式化第二个分区为 ext3 格式

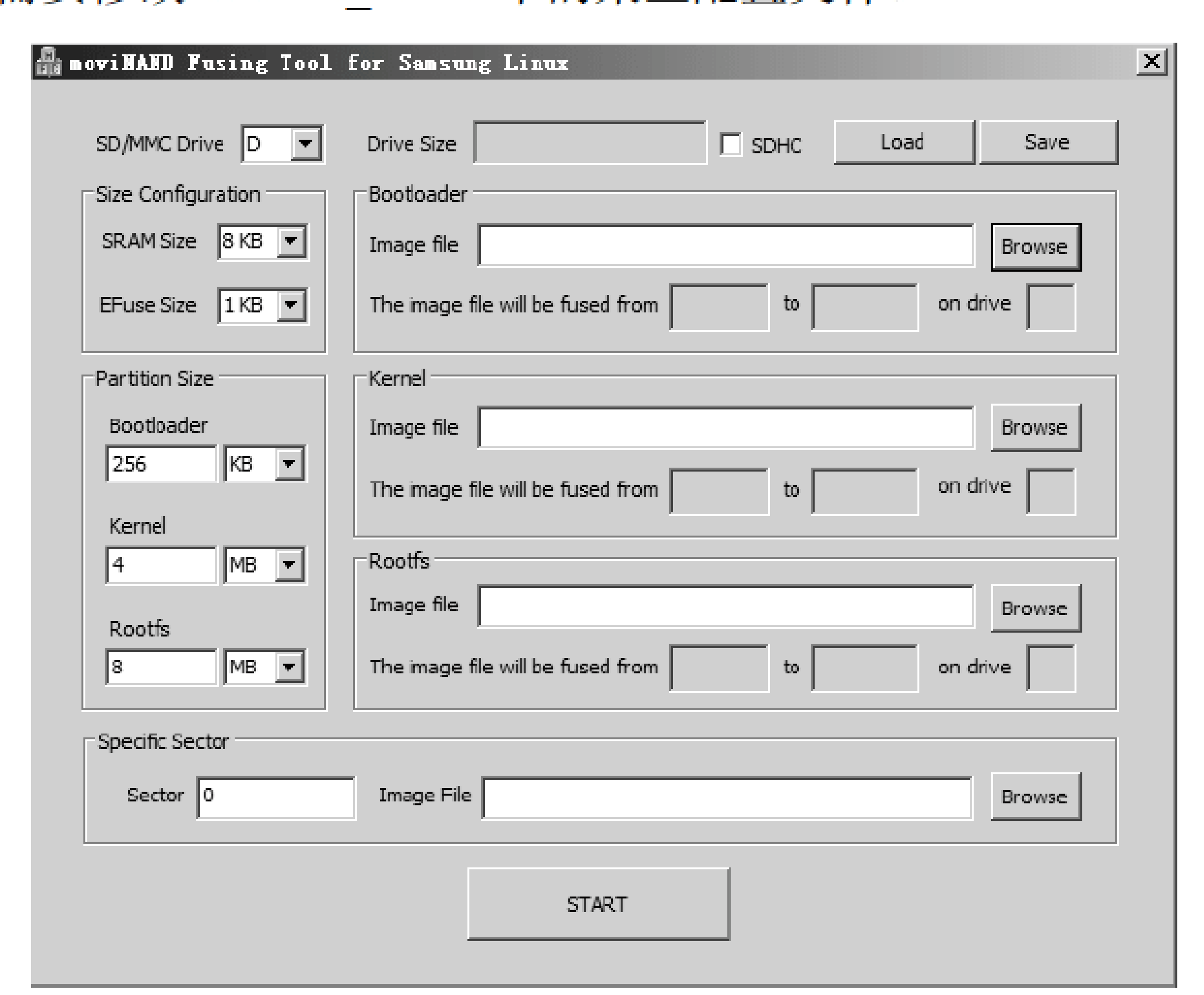


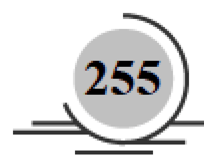
图 8-21 SD_Fusing_Tool.exe 工具

（12）断电设置开发板从 SD 卡启动，准备烧写工作。OK6410 开发板的启动规则如图 8-22 所示。

SW2 引脚号	Pin 8	Pin 7	Pin 6	Pin 5	Pin 4	Pin 3	Pin 2	Pin 1
引脚定义	SELNAND	OM4	OM3	OM2	OM1	GPN15	GPN14	GPN13
Nandflash 启动	1	0	0	1	1	X	X	X
Norflash 启动	X	0	1	0	1	X	X	X
SD 卡启动	**X**	**1**	**1**	**1**	**1**	**0**	**0**	**0**

图 8-22 OK6410 开发板的启动规则

（13）因为当前系统不是 Android，所以需要先擦除 NandFlash。运行 SD 卡中的 Boot 和 Linux，等待



30 秒钟进入 Linux 系统，如图 8-23 所示。

```
DNW v0.60C - For WinCE   [COM1,115200bps][USB:x][ADDR:0x57e00000]
Serial Port  USB Port  Configuration  Help
udevd-event[3010]: rmdir(//dev) failed: Device or resource busy


udevd-event[3011]: rmdir(//dev) failed: Device or resource busy


Gave up waiting for root device.  Common problems:
 - Boot args (cat /proc/cmdline)
   - Check rootdelay= (did the system wait long enough?)
   - Check root= (did the system wait for the right device?)
 - Missing modules (cat /proc/modules; ls /dev)
udevd-event[3019]: rmdir(//dev) failed: Device or resource busy

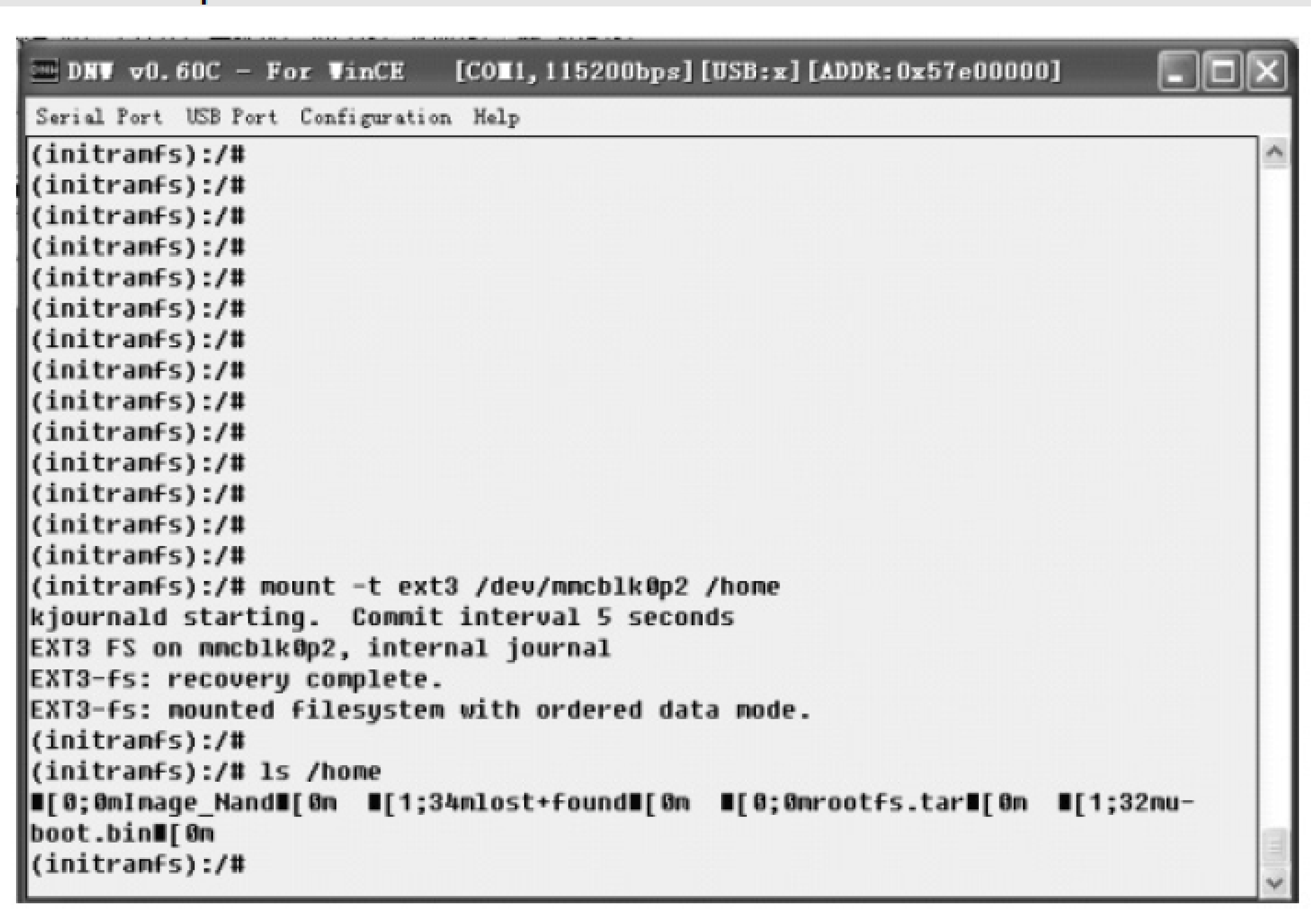

udevd-event[3020]: rmdir(//dev) failed: Device or resource busy


modprobe: chdir(2.6.29.1): No such file or directory
modprobe: chdir(2.6.29.1): No such file or directory
ALERT! /dev/mtdblock2 does not exist. Dropping to a shell!
/bin/sh: can't access tty; job control turned off
(initramfs):/#
(initramfs):/#
(initramfs):/#
```

图 8-23　进入 Linux 系统

（14）通过如下命令手动加载 SD 卡的 ext3 分区，如图 8-24 所示。

```
mount -t ext3 /dev/mmcblk0p2 /home
```



```
DNW v0.60C - For WinCE   [COM1,115200bps][USB:x][ADDR:0x57e00000]
Serial Port  USB Port  Configuration  Help
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/#
(initramfs):/# mount -t ext3 /dev/mmcblk0p2 /home
kjournald starting.  Commit interval 5 seconds
EXT3 FS on mmcblk0p2, internal journal
EXT3-fs: recovery complete.
EXT3-fs: mounted filesystem with ordered data mode.
(initramfs):/#
(initramfs):/# ls /home
■[0;0mImage_Nand■[0m  ■[1;34mlost+found■[0m  ■[0;0mrootfs.tar■[0m  ■[1;32mu-
boot.bin■[0m
(initramfs):/#
```

图 8-24　手动加载 SD 卡的 ext3 分区

（15）输入下面的命令开始烧写 u-boot。

```
flash_eraseall /dev/mtd0
```

然后输入下面的命令。

```
flashcp -v /home/u-boot.bin /dev/mtd0
```

（16）通过下面的命令烧写 Kernel。

```
flash_eraseall /dev/mtd1
```

然后输入下面的命令。

```
flashcp -v /home/Image_Nand /dev/mtd1
```

通过下面的命令烧写 Android 文件系统。

```
flash_eraseall /dev/mtd2
```

8.2　其他开发环境介绍

除了 S3C6410 平台外，还有多个常用的开发平台，本节将讲解其他几种常见平台开发环境的搭建知识。

8.2.1　基于 Cortex-A8 的 DMA-210XP 开发板

ARM Cortex-A8 处理器是第一款基于 ARMv7 架构的应用处理器，并且是有史以来 ARM 开发的性能最高、最具功率效率的处理器。Cortex-A8 处理器的速率可以在 600MHz 到超过 1GHz 的范围内调节，能够满足那些需要工作在 300mW 以下的功耗优化的移动设备的要求，并且满足那些需要 2000 Dhrystone MIPS 的性能优化的消费类应用的要求。

Cortex-A8 处理器是 ARM 的第一款超标量处理器，具有提高代码密度和性能的技术，用于多媒体和信号处理的 NEON™ 技术，以及用于高效地支持预编译和即时编译 Java 及其他字节码语言的 Jazelle&reg 运行时间编译目标（RCT）技术。为加快各大公司和厂商基于 Cortex-A8 处理器的产品上市，安赛卓尔电子科技推出的 Cortex-A8 工业开发板经国内多家厂商的使用，已在工业控制、医疗电子、节能环保、智能交通、能源节能、电力系统、通信系统、纺织行业、数控行业、汽车电子、工业触摸屏控制系统、机器人视觉、媒体处理无线应用、数字家电、车载设备、通信设备、网络终端等方面广泛应用。

Cortex-A8 处理器具备出色的运行速率和功率效率，这是通过新的支持并实现了高级泄露控制的 ARM Artisan/reg Advantage-CE 库实现的。这种处理器得到了各种各样适用于快速系统设计的 ARM 技术支持，其中包括如下 4 个方面。

- ☑　RealView&reg DEVELOP 系列软件开发工具。
- ☑　RealView CREATE 系列 ESL 工具和模型。
- ☑　CoreSight™ 调试和跟踪技术及通过 OpenMAX 多媒体处理标准实现的软件库支持。
- ☑　AMBA&reg 3 AXI 高性能 SoC 互连。

DMA-210XP 是一款基于 ARM Cortex A8 处理器的整合平台，采用 Samsung S5PV210 处理器，如图 8-25 所示。



图 8-25　DMA-210XP 开发板

Samsung S5PV210 处理器采用先进的 ARM Cortex A8 核心，运算速度可达到 1GHz，并且自带有 32/32KB 数据/指令一级缓存，512KB 二级缓存。该处理器内部集成了多媒体编解码核心（MFC），可以编解码多种格式，包括 MPEG4/H.263/H.264，并支持 VC1 解码。有了这种硬件编解码器就可以实现即时视频会议系统，类比以及 HDMI 数字视频信号输出，强大的图形处理能力能支持 1080P 高清格式以 30fps 的帧速进行高清视频重播/录制。内建的 HDMI1.3 接口能将高清影像输出至外部显示器上。另外，内部集成 3D 图形加速器（平均每秒可生成 2 千万个三角形），可以很好地支持 3D 游戏以及立体图像动态生成，可以感受到非常出色的多媒体体验。

S5PV210 为手持设备应用而设计，所以充分考虑了手持设备的低功耗要求，通过各式各样的低功率技

术，包括使用了 45 纳米（nm）低功率制程以及精细的低功率架构，DVFS 技术，降低内核工作电压等方法，确保有更长的电池使用寿命。S5PV210 处理器采用 0.65mmpitch 值的 17×17 平方毫米 FBGA 封装，降低 PCB 加工工艺要求。

另外，理器功能强大并且集成了非常丰富的外部界面，可以设计与适用于无线通信、个人导航、摄像、移动游戏、移动音乐和视频的播放、移动电视、PDA 功能、医疗器械等功能产品。

DMA-210XP 支持 Android 系统中的如下功能。

☑ 2D、3D 图形硬件加速器。
☑ MPEG4/H.263/H.264 的硬件编码与解码。
☑ 电容式多点触摸板（IIC 接口）。
☑ SDIO WiFi/USB WiFi 无线上网。
☑ 蓝牙 BT2.0+EDR 传输功能。
☑ 3G Modem 通话、上网、短信功能。
☑ CMOS 300 万像素拍照。
☑ GSensor 三轴加速器功能。
☑ HDMI 1.3，1080p 高清输出。
☑ GPS 配合 Google MAP 定位及导航使用。

8.2.2 基于 Cortex-A8 的 QT210 开发板

QT210 的 CPU 处理器是 S5PV210，这是一个 32 位低成本、低功耗、高性能的移动处理器，基于 Cortex-A8 核心 ARM V7-A 体系，运行主频 1GHz。S5PV210 内部采用 64 位总线架构，内置强大的硬件加速的视频处理器、显示控制器以及多格式音视频编解码器；具有 32K 数据缓存和 32K 指令缓存，512K L2 缓存；内置了 PowerVR SGX540 高性能图形引擎，可支持 1080p@30fps 硬件解码视频流畅播放，格式可为 MPEG4、H.263、H.264 等，最高可支持 1080p@30fps 硬件编码（MPEG-2/VC1）的视频输入。

QT210 的标准接口资源如图 8-26 所示。

QT210 的标准接口资源如下。

☑ 5V 直流电源输入接口。
☑ 两个 DB9 式 RS232 三线串口（另有两个 TTL 电平串口）。
☑ 一个 mini USB 2.0 高速 OTG 接口，支持 480Mbps。
☑ 4 路 USB 2.0 高速 Host 接口，支持 480Mbps。
☑ 1 路 3.5mm 立体声音频输出接口，1 路板载麦克风，1 路外接喇叭接口座（可直接驱动 8Ω、2W 喇叭）。
☑ 1 路标准 CVBS 电视视频输出。
☑ 1 路标准 HDMI 高清视频输出。
☑ 4 个用户 LED（红色）。
☑ 9 个侧立按键（采用矩阵键盘部分按键，鲜明标注了对应于 Android 的功能）。
☑ 一个 I2C 三轴重力加速传感器。
☑ 一个复位按钮。
☑ 两个 micro SD（TF）卡座。
☑ 板载实时时钟备份电池。
☑ 一个 JTAG 接口，20pin 2.0mm 间距。
☑ 一个 CMOS 摄像头接口，20pin 2.0mm 间距。

☑　一个矩阵键盘接口，20pin 2.0mm 间距，可连接使用 8×8 矩阵键盘。
☑　一个 GPIO 接口，20pin 2.0mm 间距，包含 AD 输入、中断引脚、SPI 和 UART 等端口。

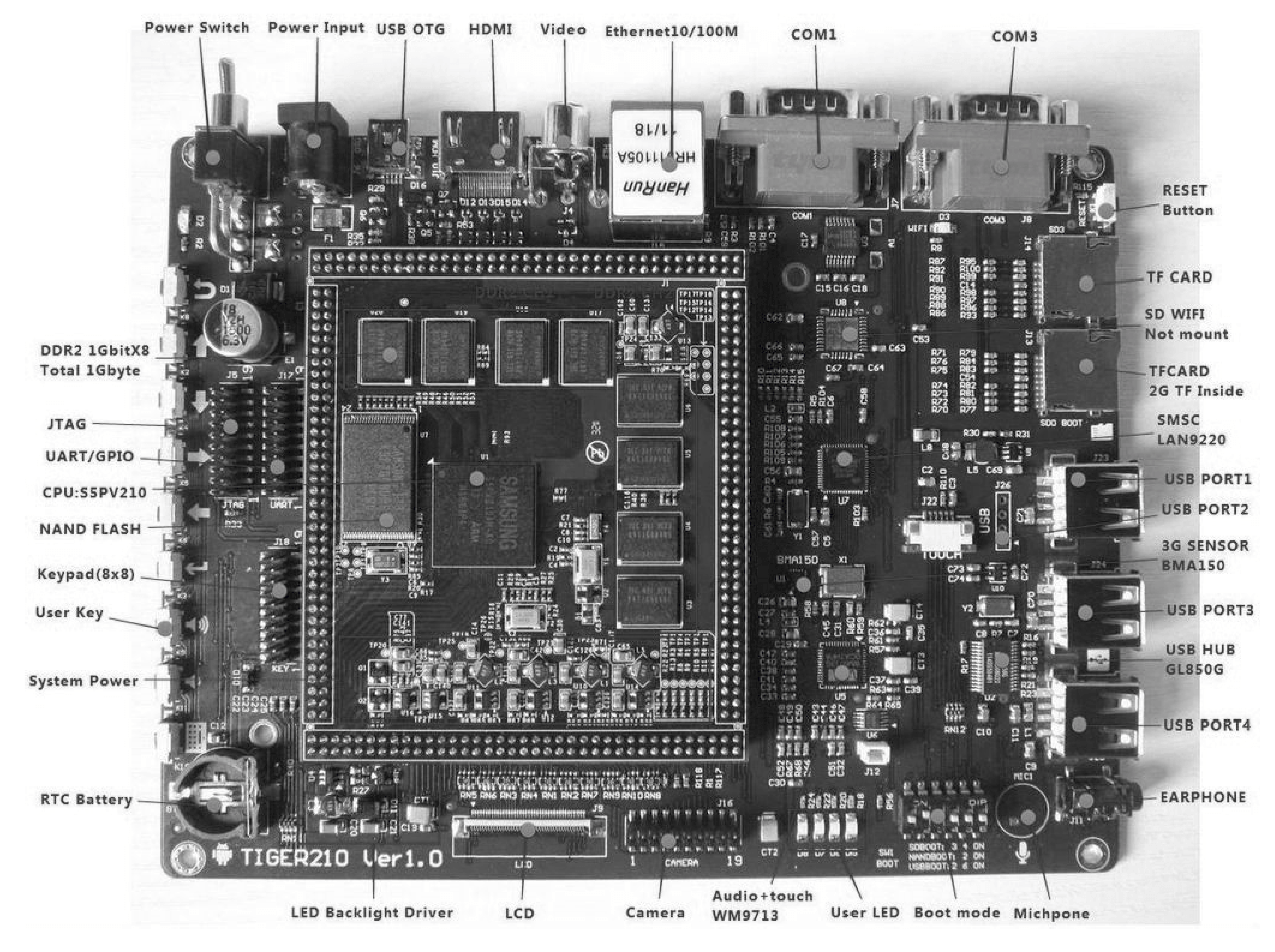


图 8-26　QT210 的标准接口

8.2.3　X210CV3 开发板

X210CV3 是九鼎创展继 X210CV01 和 X210CV02 推出的一款低功耗、高性能、可扩展性强的核心板，由深圳市九鼎创展科技设计生产并发行销售。X210CV3 采用三星 Cortex-A8 架构的 S5PV210 作为主处理器，运行速度高达 1GHz。PCB 采用 8 层沉金工艺设计，具有最佳的电气特性和抗干扰特性，可稳定工作于 1GHz，和 X210CV01、X210CV02 相比，其接口更加齐全，现已经被广泛应用于 MID、POS、PDA、PND、智能家居、智能刷卡终端、考试设备、手机、学习机、船舶、医疗等各种行业的工控领域。

S5PV210 内部集成了 PowerVR SGX540 的高性能图形引擎，支持 3D 图形流畅运行，可以流畅编解码 1080P 的视频文件。S5PV210 出色的性能，配合 x210v3 底板，能够完美展现芯片的大多数功能，可以大大缩短用户的开发周期。

8.3　测试驱动的方法

在编写驱动程序完毕后及在完成移植驱动程序的工作后，需要测试这个驱动程序的正确性和完整性。假设编写驱动程序文件 word_tongji.c，功能是从/dev/word_tongji 设备中读取文件的数据，具体实现代码如下

所示。

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <asm/uaccess.h>

#define DEVICE_NAME "wordcount"
static unsigned char mem[10000];
static int word_tongji = 0;
#define TRUE 255
#define FALSE 0

static unsigned char is_spacewhite(char c)
{
	if (c == 32 || c == 9 || c == 13 || c == 10)
		return TRUE;
	else
		return FALSE;
}

static int get_word_tongji(const char *buf)
{
	int n = 1;
	int i = 0;
	char c = ' ';

	char flag = 0;
	if (*buf == '\0')
		return 0;
	if (is_spacewhite(*buf) == TRUE)
		n--;
	for (; (c = *(buf + i)) != '\0'; i++)
	{
		if (flag == 1 && is_spacewhite(c) == FALSE)
		{
			flag = 0;
		}
		else if (flag == 1 && is_spacewhite(c) == TRUE)
		{
			continue;
		}
		if (is_spacewhite(c) == TRUE)
		{
			n++;
			flag = 1;
		}
	}
	if (is_spacewhite(*(buf + i - 1)) == TRUE)
		n--;
```

```
        return n;
}

static ssize_t word_tongji_read(struct file *file, char __user *buf,
            size_t count, loff_t *ppos)
{
        unsigned char temp[4];
        temp[0] = word_tongji >> 24;
        temp[1] = word_tongji >> 16;
        temp[2] = word_tongji >> 8;
        temp[3] = word_tongji;
        if (copy_to_user(buf, (void*) temp, 4))
        {
                return -EINVAL;
        }
        printk("read:word tongji:%d", (int) count);
        return count;
}

static ssize_t word_tongji_write(struct file *file, const char __user *buf,
            size_t count, loff_t *ppos)
{
        ssize_t written = count;
        if (copy_from_user(mem, buf, count))
        {
                return -EINVAL;
        }
        mem[count] = '\0';
        word_tongji = get_word_tongji(mem);
        printk("write:word tongji:%d\n", (int) word_tongji);
        return written;
}

static struct file_operations dev_fops =
{ .owner = THIS_MODULE, .read = word_tongji_read, .write = word_tongji_write };
static struct miscdevice misc =
{ .minor = MISC_DYNAMIC_MINOR, .name = DEVICE_NAME, .fops = &dev_fops };
static int __init word_tongji_init(void)
{
        int ret;
        ret = misc_register(&misc);
        printk("word_tongji_init_success\n");
        return ret;
}
static void __exit word_tongji_exit(void)
{
        misc_deregister(&misc);
        printk("word_tongji_init_exit_success\n");
}

module_init( word_tongji_init);
```

```
module_exit( word_tongji_exit);
MODULE_AUTHOR("lining");
MODULE_DESCRIPTION("statistics of word tongji.");
MODULE_ALIAS("word tongji module.");
MODULE_LICENSE("GPL");
```

下面将详细讲解测试底层驱动程序的方法。

8.3.1 使用 Ubuntu Linux 测试驱动

在本节前面编写的驱动程序文件 word_tongji.c 中，实现了 Linux 驱动程序通过 4 个字节从设备文件（/dev/wordcount）返回单词数的功能。虽然不能使用 cat 命令测试驱动程序（cat 命令不会将这 4 个字节还原成 int 类型的值显示），但是可以从日志中使用如下命令查看单词数。

```
# sh build.sh
# echo 'what is your name a'   > /dev/wordcount
# dmesg
```

执行上面的命令后，驱动程序成功统计单词数后输出如图 8-27 所示的信息。

```
[799954.766023] word_count_init_exit_success
[799954.769331] word_count_init_success
[799981.133047] word_count_init_exit_success
[799981.136687] word_count_init_success
[799993.040473] word_count_init_exit_success
[799993.046582] word_count_init_success
[800010.729989] write:word count:5
root@vm-ubuntu:~/drivers/ch06/word_count#
```

图 8-27 输出统计的字符数

在上述测试过程中，虽然使用 echo 和 dmesg 命令可以测试 Linux 驱动程序，但是为了使测试效果更接近真实环境，在现实中通常需要编写专门测试程序。例如我们为 word_tongji 驱动编写一个专门的测试程序 test_word_tongji.c，文件 test_word_tongji.c 的功能是通过直接操作/dev/wordcount 设备文件的方式与 word_tongji 驱动进行交互。测试文件 test_word_tongji.c 的具体实现代码如下所示。

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
	int testdev;
	unsigned char buf[4];

	testdev = open("/dev/wordcount", O_RDWR);
	if (testdev == -1)
	{
		printf("Cann't open file \n");
		return 0;
	}
	if (argc > 1)
	{

		write(testdev, argv[1], strlen(argv[1]));
		printf("string:%s\n", argv[1]);
```

```
    }

    read(testdev, buf, 4);

    int n = 0;
    n = ((int) buf[0]) << 24 | ((int) buf[1]) << 16 | ((int) buf[2]) << 8
            | ((int) buf[3]);
    printf("display word byte :%d,%d,%d,%d\n", buf[0], buf[1], buf[2], buf[3]);
    printf("word tongji:%d\n", n);
    close(testdev);
    return 0;
}
```

上述 test_word_tongji 程序代码可以跟一个命令行参数。如果在命令行参数值中含有空格符，则需要使用单引号（'）或双引号（"）将参数值括起来。例如可以使用下面的命令来测试 word_tongji 驱动程序。

```
# gcc test_word_tongji.c   -o test_word_tongji
# test_word_tongji
# test_word_count "what is your name a."
```

执行上面的命令后会输出如图 8-28 所示的信息，表示 word_count 驱动测试成功。



图 8-28　输出测试信息

8.3.2　在 Android 模拟器中测试驱动

在 Android 模拟器中，可以通过原生（Native）的 C 程序来测试 Linux 驱动。在使用这种测试方式之前，需要先在模拟器上安装 word_count.ko 驱动模块。假如将 word_tongji.ko 驱动模块直接安装在 Android 模拟器中，具体实现流程如下。

（1）执行 build.sh 脚本，并选择“Android 模拟器”。

（2）脚本会自动将 word_tongji.ko 文件上传到 Android 模拟器的/data/local 目录中，并进行安装。

注意：如果使用的是 S3C6410 开发板，在安装 word_tongji.ko 文件时就会输出如下错误信息：

```
insmod:init_module '/data/local/word_count.ko' failed（Function not implemented）
```

这个错误表示编译 Linux 驱动的 Linux 内核版本与当前 Android 模拟器的版本不相同，这样将无法安装。由此可见，在编译 Linux 驱动时必须选择与当前运行的 Linux 内核版本相同的 Linux 内核进行编译，否则就不能成功安装 Linux 驱动。

因为在 Android 模拟器中，其 Goldfish 内核默认不允许动态装载 Linux 驱动模块，所以在编译 Linux 内核前需要执行如下配置 Linux 内核的命令。

```
# cd ~/kernel/goldfish
# make menuconfig
```

执行上面的命令后弹出如图 8-29 所示的设置界面。

按下空格键，选中 Enable loadable module support（前面是[*]）选项，然后回车进入到子菜单，并选中前面的 3 个选项，如图 8-30 所示，否则 Linux 驱动模块仍然无法安装和卸载。

退出设置菜单时需要保持当前的设置，接下来需要重新编译 Linux 内核。成功编译内核后，在模拟器中可以使用新生成的 zImage 内核文件动态装载 Linux 驱动模块。此时执行 build.sh 脚本文件可以完成对 word_count 驱动的编译、上传和安装工作，进入 Android 模拟器终端可以使用 echo 和 dmesg 命令测试

word_count 驱动并查看测试结果。

```
.config - Linux Kernel v2.6.29 Configuration
                    Linux Kernel Configuration
Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters
are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press
<Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]
excluded  <M> module  < > module capable

          General setup  --->
      [*] Enable loadable module support  --->
      -*- Enable the block layer  --->
          System Type  --->
          Bus support  --->
          Kernel Features  --->
          Boot options  --->
          CPU Power Management  --->
          Floating point emulation  --->
          Userspace binary formats  --->
          Power management options  --->
      [*] Networking support  --->
          Device Drivers  --->
          File systems  --->
      v(+)

              <Select>    < Exit >    < Help >
```

图 8-29　Linux 内核设置菜单

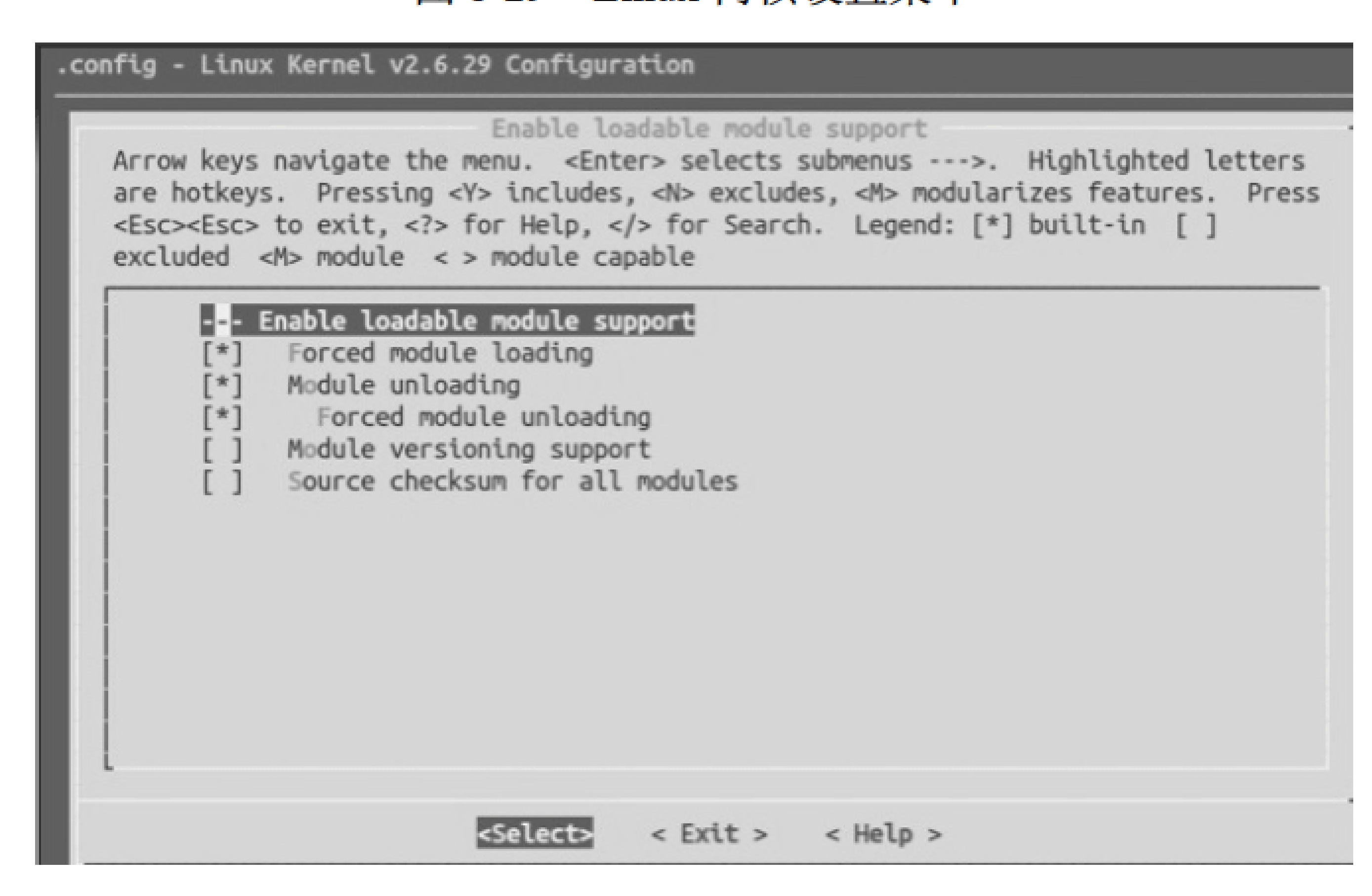


图 8-30　设置子菜单

接下来看编译 test_word_tongji.c 文件的过程，使用 Android.mk 设置编译参数，并使用 make 命令进行编译。首先在 test_word_tongji.c 的同一个目录中建立一个 Android.mk 文件，并输入如下内容。

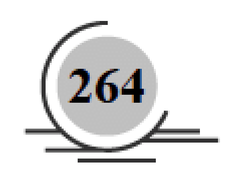
```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
# 指定要编译的源代码文件
LOCAL_SRC_FILES:= test_word_tongji.c
# 指定模块名，也是编译后生成的可执行文件名
LOCAL_MODULE := test_word_count
LOCAL_MODULE_TAGS := optional
include $(BUILD_EXECUTABLE)
```

在 Android.mk 文件中，LOCAL_MODULE_TAGS 表示当前工程（Android.mk 文件所在的目录）在什么模式下编译。如果设置为 optional，则表示不考虑模式，在任何模式下都能编译。该变量可以设置的值有 user、userdebug、eng、optional，具体说明如下。

☑　user：限制用户对 Android 系统的访问，适合于发布产品。

☑　userdebug：类似于 user 模式，但拥有 root 访问权限，并且可以从日志中获取大量的调试信息。

☑　eng：这是默认值，一般在开发的过程中设置该模式。除了拥有 userdebug 的全部功能外，还会带



有大量的调试工具。

LOCAL_MODULE_TAGS 的值与 TARGET_BUILD_VARIANT 变量有关。TARGET_BUILD_VARIANT 变量用于设置当前的编译模式，可设置的值包括 user、userdebug 和 eng。如果想改变编译模式，可以在编译 Android 源代码之前执行如下命令。

```
# export TARGET_BUILD_VARIANT = user
```

或者使用 lunch 命令设置编译模式。

```
# lunch full-eng
```

在上述指令中，full 表示要建立的目标，除了 full 目标（为所有的平台建立）外，还有专门为 x86 建立的 full-x86。

在 Android.mk 文件中，BUILD_EXECUTABLE 表示建立可执行的文件。可执行文件路径是<Android 源代码目录>/out/target/product/generic/system/bin/test_word_tongji。可以使用 include $(BUILD_SHARED_LIBRARY)编译成动态库（.so）文件，动态库的路径是<Android 源代码目录>/out/target/product/generic/system/lib/test_word_tongji.so。可以使用 include $(BUILD_STATIC_LIBRARY)编译成静态库（.a）文件，静态库的路径是<Android 源代码目录>/out/target/product/generic/obj/STATIC_LIBRARIES/test_word_tongji_intermediates/test_word_tongji。

为了将文件 test_word_tongji.c 编译成可以在 Android 模拟器上运行的可执行程序，需要将 word_tongji 目录复制到<Android 源代码目录>的某个子目录中，也可以在<Android 源代码目录>目录中为 word_tongji 目录建立一个符号链接。假设 Android 源代码的目录是/sources/android/android4/development/word_tongji，可以使用如下命令为 word_tongji 目录在<Android 源代码目录>/development 目录下建立一个符号链接。

```
# ln -s   word_tongji /sources/android/android4/development/word_tongji
```

进入/sources/android/android4 目录，执行下面的命令初始化编译命令。

```
# source ./build/envsetup.sh
```

在编译过程中，可以使用下面两种方法来编译 test_word_tongji.c 文件。

☑　进入/sources/android/android4/development/word_tongji 目录，并执行如下的命令。

```
# mm
```

☑　在/sources/android/android4 目录下执行如下的命令。

```
# mmm   development/word_tongji
```

成功编译后，可以在<Android 源代码目录>/out/target/product/generic/system/bin 目录下找到文件 test_word_tongji。

运行下面的命令，目的是将文件 test_word_tongji 上传到 Android 模拟器。

```
# adb push   ./emulator/test_word_tongji/data/local
```

接下来进入 Android 模拟器的终端，并执行下面的命令测试 word_tongji 驱动。注意，这一步的前提是先使用 chmod 命令设置 test_word_tongji 的可执行权限。

```
# chmod 777 /data/local/test_word_tongji
# /data/local/test_word_tongji
# /data/local/test_word_count   'what is your name a'
```

执行上面的命令后输出的单词个数是 5，这表示成功测试了我们的驱动程序。

注意： 在 Android 模拟器中，不仅可以使用 Linux 命令测试驱动，也可以像 UbuntuLinux 一样使用本地 C/C++ 程序进行测试。要想在模拟器中直接运行普通的 Linux 程序，需要事先满足如下两个条件。

（1）无论是 Android 模拟器，还是开发板或手机，都需要有 root 权限。

（2）需要使用交叉编译器编译可执行文件，以便支持 ARM 处理器。

第 9 章　低内存管理驱动

在 Android 系统中，Low Memory Killer（低内存管理）驱动在用户空间中指定了一组内存临界值，当其中某个值与进程描述中的 oom_adj 值在同一范围时，该进程将被 Kill（杀）掉（在 parameters/adj 中指定 oome_adj 的最小值）。在 Android 系统中，其 Low Memory Killer 驱动基于 Linux 系统的 OOM 系统。本章将详细讲解 Android 系统中 Low Memory Killer 驱动的基本知识，分析其具体架构原理和实现源码。

9.1　OOM 机制

内存泄漏也称作“存储渗漏”，是指用动态存储分配函数动态开辟的空间，但是在使用完毕后没有进行及时释放工作，这样会导致一直占据该内存单元的结果发生并直到程序结束。内存泄漏问题一直是影响智能设备处理速度的最大因素之一。本节将详细分析 Linux 系统中 OOM 机制的基本知识。

9.1.1　OOM 机制基础

在传统的 PC 机系统中，如果某个程序发生了内存泄漏，系统通常会将其进程 Kill 掉。在 Linux 系统中，使用了一种名为 OOM（Out Of Memory，内存不足）的机制来完成这个任务，该机制会在系统内存不足的情况下选择一个进程并将其 Kill 掉。

在 Linux 系统中，OOM killer（Out Of Memory killer）机制会监控那些占用内存过大的进程，例如在瞬间占用大内存的进程。为了防止发生内存耗尽的情况，此机制会自动杀掉该进程。要想预防重要系统进程不小心触发（OOM）机制而被杀死，可以设置/proc/PID/oom_adj 参数为-17，并临时关闭 Linux 内核的 OOM 机制。这样系统内核会使用某算法给每个进程计算一个分数，通过这个分数来决定杀死哪个进程。具体每个进程的 OOM 分数，可以从/proc/PID/oom_score 中得到。

如果想保护某个进程不被内核杀掉，可以通过如下指令进行操作。

```
echo -17 > /proc/$PID/oom_adj
```

如果想防止 sshd 进程被杀死，可以通过如下指令进行操作。

```
pgrep -f "/usr/sbin/sshd" | while read PID;do echo -17 > /proc/$PID/oom_adj;done
```

为了确保万无一失，可以在计划任务中加入如下的定时任务指令。

```
#/etc/cron.d/oom_disable
*/1**** root pgrep -f "/usr/sbin/sshd" | while read PID;do echo -17 > /proc/$PID/oom_adj;done
```

为了避免发生重启失效的问题，可以写入/etc/rc.d/rc.local。

```
echo -17 > /proc/$(pidof sshd)/oom_adj
```

注意： 在上述设置指令操作的过程中，使用-17 而不是其他数值的原因是由 Linux 内核定义决定的。通过查看内核源码（以 Linux-3.4 版本的 kernel 源码为例，其路径为 linux-3.4/include/linux/oom.h）可知，oom_adj 的可调值为-16～15，其中 15 最大，-16 最小，-17 为禁止使用 OOM。oom_score 的值是通过 2 的 n 次方计算出来的，其中 n 就是进程的 oom_adj 值，所以 oom_score 的分数越高就越会被内核

```
优先杀掉。
#define OOM_DISABLE (-17)
/* inclusive */
#define OOM_ADJUST_MIN (-16)
#define OOM_ADJUST_MAX 15
```

除此之外，还可以通过修改内核参数的方式来禁止 OOM 机制，具体命令如下所示。

```
# sysctl -w vm.panic_on_oom=1
vm.panic_on_oom = 1 //1 表示关闭，默认为 0 表示开启 OOM
# sysctl –p
```

9.1.2　分析 OOM 机制的具体实现

在 Linux 系统中，OOM 机制的实现文件是 mm/oom_kill，文件 oom_kill.c 的具体实现流程如下。

（1）在系统中分配内存时会进行判断处理，当内存不足时会调用函数 out_of_memory()进行处理。函数 out_of_memory()的具体实现代码如下所示。

```
void out_of_memory(struct zonelist *zonelist, gfp_t gfp_mask,
                int order, nodemask_t *nodemask, bool force_kill)
{
        const nodemask_t *mpol_mask;
        struct task_struct *p;
        unsigned long totalpages;
        unsigned long freed = 0;
        unsigned int uninitialized_var(points);
        enum oom_constraint constraint = CONSTRAINT_NONE;
        int killed = 0;

        blocking_notifier_call_chain(&oom_notify_list, 0, &freed);
        if (freed > 0)
                /* Got some memory back in the last second. */
                return;

        /*
         * If current has a pending SIGKILL or is exiting, then automatically
         * select it.  The goal is to allow it to allocate so that it may
         * quickly exit and free its memory.
         */
        if (fatal_signal_pending(current) || current->flags & PF_EXITING) {
                set_thread_flag(TIF_MEMDIE);
                return;
        }

        /*
         * Check if there were limitations on the allocation (only relevant for
         * NUMA) that may require different handling.
         */
        constraint = constrained_alloc(zonelist, gfp_mask, nodemask,
                                                &totalpages);
        mpol_mask = (constraint == CONSTRAINT_MEMORY_POLICY) ? nodemask : NULL;
```

```
        check_panic_on_oom(constraint, gfp_mask, order, mpol_mask);

        if (sysctl_oom_kill_allocating_task && current->mm &&
            !oom_unkillable_task(current, NULL, nodemask) &&
            current->signal->oom_score_adj != OOM_SCORE_ADJ_MIN) {
                get_task_struct(current);
                oom_kill_process(current, gfp_mask, order, 0, totalpages, NULL,
                                 nodemask,
                                 "Out of memory (oom_kill_allocating_task)");
                goto out;
        }

        p = select_bad_process(&points, totalpages, mpol_mask, force_kill);
        /* Found nothing?!?! Either we hang forever, or we panic. */
        if (!p) {
                dump_header(NULL, gfp_mask, order, NULL, mpol_mask);
                panic("Out of memory and no killable processes...\n");
        }
        if (p != (void *)-1UL) {
                oom_kill_process(p, gfp_mask, order, points, totalpages, NULL,
                                 nodemask, "Out of memory");
                killed = 1;
        }
out:
        /*
         * Give the killed threads a good chance of exiting before trying to
         * allocate memory again.
         */
        if (killed)
                schedule_timeout_killable(1);
}
```

（2）在上述代码中调用了函数 select_bad_process()，功能是根据当前运行 task（进程）的内存，参照 oom score 信息得到 point 值最高的那一个。函数 select_bad_process()的具体实现代码如下所示。

```
static struct task_struct *select_bad_process(unsigned int *ppoints,
        unsigned long totalpages, struct mem_cgroup *memcg,
        const nodemask_t *nodemask, bool force_kill)
{
    struct task_struct *g, *p;
    struct task_struct *chosen = NULL;
    *ppoints = 0;
    /*遍历所有进程*/
    do_each_thread(g, p) {
        unsigned int points;
        /*不理会处于退出的进程*/
        if (p->exit_state)
            continue;
        /*不能杀掉 init、kernel_thread 等核心的线程*/
        if (oom_unkillable_task(p, memcg, nodemask))
            continue;
        /*不理会正在被 oom killing 的进程*/
```

```
            if (test_tsk_thread_flag(p, TIF_MEMDIE)) {
                if (unlikely(frozen(p)))
                    __thaw_task(p);
                if (!force_kill)
                    return ERR_PTR(-1UL);
            }
            if (!p->mm)
                continue;
            if (p->flags & PF_EXITING) {
                if (p == current) {
                    chosen = p;
                    *ppoints = 1000;
                } else if (!force_kill) {
                    /*
                     * If this task is not being ptraced on exit,
                     * then wait for it to finish before killing
                     * some other task unnecessarily.
                     */
                    if (!(p->group_leader->ptrace & PT_TRACE_EXIT))
                        return ERR_PTR(-1UL);
                }
            }
            /*计算 task 对应的 points*/
            points = oom_badness(p, memcg, nodemask, totalpages);
            /*如果这个 task 比上一次的 points 大，则保存 point*/
            if (points > *ppoints) {
                chosen = p;
                *ppoints = points;
            }
        } while_each_thread(g, p);
        return chosen;
}
```

（3）函数 oom_badness()的功能是计算 task 对应的 points 值，具体实现代码如下所示。

```
unsigned int oom_badness(struct task_struct *p, struct mem_cgroup *memcg,
                    const nodemask_t *nodemask, unsigned long totalpages)
{
        long points;
        if (oom_unkillable_task(p, memcg, nodemask))
            return 0;
        p = find_lock_task_mm(p);
        if (!p)
            return 0;
        /*不处理 oom_score_adj 为-1000 的，此值可以通过/proc/pid_num/oom_score_adj 设置*/
        /*设置范围为-1000 ~ 1000，值越大越容易被 oom kill 掉*/
        if (p->signal->oom_score_adj == OOM_SCORE_ADJ_MIN) {
            task_unlock(p);
            return 0;
        }
        /*
         * The memory controller may have a limit of 0 bytes, so avoid a divide
         * by zero, if necessary.
```

```
	 */
	if (!totalpages)
		totalpages = 1;
	/* get_mm_rss 获取当前用户空间使用文件和匿名页占有内存数，nr_ptes 获取
当前保存页表使用的内存*/
	points = get_mm_rss(p->mm) + p->mm->nr_ptes;
	/*获取交换内存使用的内存数*/
	points += get_mm_counter(p->mm, MM_SWAPENTS);
	/*每个 task 同等计算，可不管*/
	points *= 1000;
	points /= totalpages;
	task_unlock(p);
	/*当该进程具有 CAP_SYS_ADMIN 能力，那么 Point 降低，因为具有 ADMIN 权限的
Task 是被认为表现良好的 */
	if (has_capability_noaudit(p, CAP_SYS_ADMIN))
		points -= 30;

	/*加上 oom_score_adj，范围从-1000 ~ 1000 */
	points += p->signal->oom_score_adj;

	/*
	 * Never return 0 for an eligible task that may be killed since it's
	 * possible that no single user task uses more than 0.1% of memory and
	 * no single admin tasks uses more than 3.0%.
	 */
	if (points <= 0)
		return 1;
	/*1000 封顶*/
	return (points < 1000) ? points : 1000;
}
```

（4）函数 oom_kill_process()的功能是执行具体的杀死进程操作，即杀死一个指定的进程。函数 oom_kill_process()的具体实现代码如下所示。

```
void oom_kill_process(struct task_struct *p, gfp_t gfp_mask, int order,
			unsigned int points, unsigned long totalpages,
			struct mem_cgroup *memcg, nodemask_t *nodemask,
			const char *message)
{
	struct task_struct *victim = p;
	struct task_struct *child;
	struct task_struct *t = p;
	struct mm_struct *mm;
	unsigned int victim_points = 0;
	static DEFINE_RATELIMIT_STATE(oom_rs, DEFAULT_RATELIMIT_INTERVAL,
					DEFAULT_RATELIMIT_BURST);

	/*
	 * If the task is already exiting, don't alarm the sysadmin or kill
	 * its children or threads, just set TIF_MEMDIE so it can die quickly
	 */
	if (p->flags & PF_EXITING) {
```

```
			set_tsk_thread_flag(p, TIF_MEMDIE);
			put_task_struct(p);
			return;
		}

		if (__ratelimit(&oom_rs))
			dump_header(p, gfp_mask, order, memcg, nodemask);

		task_lock(p);
		pr_err("%s: Kill process %d (%s) score %d or sacrifice child\n",
			message, task_pid_nr(p), p->comm, points);
		task_unlock(p);

		/*
		 * If any of p's children has a different mm and is eligible for kill,
		 * the one with the highest oom_badness() score is sacrificed for its
		 * parent.   This attempts to lose the minimal amount of work done while
		 * still freeing memory.
		 */
		read_lock(&tasklist_lock);
		do {
			list_for_each_entry(child, &t->children, sibling) {
				unsigned int child_points;

				if (child->mm == p->mm)
					continue;
				/*
				 * oom_badness() returns 0 if the thread is unkillable
				 */
				child_points = oom_badness(child, memcg, nodemask,
									totalpages);
				if (child_points > victim_points) {
					put_task_struct(victim);
					victim = child;
					victim_points = child_points;
					get_task_struct(victim);
				}
			}
		} while_each_thread(p, t);
		read_unlock(&tasklist_lock);

		rcu_read_lock();
		p = find_lock_task_mm(victim);
		if (!p) {
			rcu_read_unlock();
			put_task_struct(victim);
			return;
		} else if (victim != p) {
			get_task_struct(p);
			put_task_struct(victim);
			victim = p;
```

```
471             }
472 
473             /* mm cannot safely be dereferenced after task_unlock(victim) */
474             mm = victim->mm;
475             pr_err("Killed process %d (%s) total-vm:%lukB, anon-rss:%lukB, file-rss:%lukB\n",
476                     task_pid_nr(victim), victim->comm, K(victim->mm->total_vm),
477                     K(get_mm_counter(victim->mm, MM_ANONPAGES)),
478                     K(get_mm_counter(victim->mm, MM_FILEPAGES)));
479             task_unlock(victim);
480 
481             /*
482              * Kill all user processes sharing victim->mm in other thread groups, if
483              * any.   They don't get access to memory reserves, though, to avoid
484              * depletion of all memory.   This prevents mm->mmap_sem livelock when an
485              * oom killed thread cannot exit because it requires the semaphore and
486              * its contended by another thread trying to allocate memory itself.
487              * That thread will now get access to memory reserves since it has a
488              * pending fatal signal.
489              */
490             for_each_process(p)
491                     if (p->mm == mm && !same_thread_group(p, victim) &&
492                         !(p->flags & PF_KTHREAD)) {
493                             if (p->signal->oom_score_adj == OOM_SCORE_ADJ_MIN)
494                                     continue;
495 
496                             task_lock(p);     /* Protect ->comm from prctl() */
497                             pr_err("Kill process %d (%s) sharing same memory\n",
498                                     task_pid_nr(p), p->comm);
499                             task_unlock(p);
500                             do_send_sig_info(SIGKILL, SEND_SIG_FORCED, p, true);
501                     }
502             rcu_read_unlock();
503 
504             set_tsk_thread_flag(victim, TIF_MEMDIE);
505             do_send_sig_info(SIGKILL, SEND_SIG_FORCED, victim, true);
506             put_task_struct(victim);
507 }
```

在上述代码中，在第 440 行开始的 do 循环语句中，表示当真的需要用较多内存时可能会杀掉子进程，而父进程还可以活着。在第 490 行开始的 for_each_process 循环语句中，表示只要 mm 相同则是共享内存的进程，这将和当前找到最高 point 的指定进程一起被杀掉。

（5）函数 check_panic_on_oom()的功能是检查处理当前运行的进程，在用户空间中可以通过/proc/sys/vm/panic_on_oom 值来改变 oom 的行为，其中 1 表示 oom 时直接 panic，0 表示只杀掉 best 进程而让系统继续运行。函数 check_panic_on_oom()的具体实现代码如下所示。

```
void check_panic_on_oom(enum oom_constraint constraint, gfp_t gfp_mask,
                        int order, const nodemask_t *nodemask)
{
        if (likely(!sysctl_panic_on_oom))
                return;
        if (sysctl_panic_on_oom != 2) {
                /*
```

```
                 * panic_on_oom == 1 only affects CONSTRAINT_NONE, the kernel
                 * does not panic for cpuset, mempolicy, or memcg allocation
                 * failures.
                 */
                if (constraint != CONSTRAINT_NONE)
                        return;
        }
        dump_header(NULL, gfp_mask, order, NULL, nodemask);
        panic("Out of memory: %s panic_on_oom is enabled\n",
                sysctl_panic_on_oom == 2 ? "compulsory" : "system-wide");
}
```

9.2　Android 系统的 Low Memory Killer 架构机制

在 Android 系统中，通过 Low Memory Killer 在用户空间中设置了一组内存临界值。如果里面的某个值与进程描述中的 oom_adj 值在同一个范围，则会 Kill 掉该进程。在文件/sys/module/lowmemorykiller/parameters/adj 中指定了 oom_adj 的最小值，在文件/sys/module/lowmemorykiller/parameters/minfree 中存储空闲页面的数量。

存储的空闲页面数量值都用一个逗号将其隔开且以升序排列，例如将“0,9”写入/sys/module/lowmemorykiller/parameters/adj 中，把“1024,4096”写入/sys/module/lowmemory-killer/parameters/minfree 中，就表示当一个进程的空闲存储空间下降到 4096 个页面时，会 Kill 掉 oom_adj 值为 9 的或者更大的进程。同样的道理，当一个进程的空闲存储空间下降到 1024 个页面时，会 Kill 掉 oom_adj 值为 0 或者更大的进程。其实在文件 lowmemorykiller.c 中会发现指定了这样的值，具体代码如下所示。

```
static int lowmem_adj[6] = {
    0,
    1,
    6,
    12,
};
static int lowmem_adj_size = 4;
static size_t lowmem_minfree[6] = {
    3*512, //6MB
    2*1024, //8MB
    4*1024, //16MB
    16*1024, //64MB
};
static int lowmem_minfree_size = 4;
```

由此可见，当一个进程的空闲空间在下降到 3512 个页面时，会 Kill 掉 oom_adj 值为 0 或者更大的进程；当一个进程的空闲空间下降到 21024 个页面时，会 Kill 掉 oom_adj 值为 10 或者更大的进程，继续下去，依次类推。其实在现实应用中，可以将上述过程概括为一个规律：满足如下规则的进程会被优先杀掉。

- ☑ task_struct->signal_struct->oom_adj，越大的越优先被 Kill。
- ☑ 占用物理内存最多的那个进程会被优先 Kill。

在上述规则中，signal_struct->oom_adj 表示当内存短缺时进程被选择并 Kill 的优先级，取值范围是−17~15。如果是−17，则表示不会被选中，值越大越可能被选中。当某个进程被选中后，内核会发送 SIGKILL

信号将其 Kill 掉。

实际上，Low Memory Killer 驱动程序会认为被用于缓存的存储空间都要被释放。如果很多缓存存储空间处于被锁定的状态，并且当正常的 oom killer 被触发之前不会 Kill 掉这些进程，则将是一个非常严重的错误。

注意：Low Memory Killer 机制和 OOM 的对比

Android 系统的 Low Memory Killer 机制和 Linux 标准 OOM（Out Of Memory）机制相比，Low Memory Killer 更加灵活。当内存不够时，该策略会试图结束一个进程。组件 Low Memory Killer 通过调用 Linux 内存管理系统的接口来注册一个 shrinker，此处的 shrinker 通过 Low Memory Killer 实现。

标准 Linux 内核 OOM Killer 在 mm/oom_kill.c 中实现，在 mm/page_alloc.a_alloc_pages_may_oom 中被调用。文件 oom_kill.c 最主要的函数是 out_of_memory()，它选择一个 bad 进程杀死，通过发送 SIGKILL 信号来杀死进程。

在 out_of_memory 中通过调用 select_bad_process 选择杀死一个进程，选择的依据在 badness()函数中实现，基于多个标准来给每个进程算分，分数最高的被选中杀死。基本上是占用内存越多，oom_adj 越大越有可能被选中。

由此可以看出，Android 的 Low Memory Killer 和标准的 OOM Killer 的很多思路是一致的，只不过 Low Memory Killer 作为一个 shrinker 实现；而 OOM Killer 则在分配内存时被调用(如果内存资源很紧张)。Android 的 Low Memory Killer 实现的较为简洁，这点从代码尺寸就能看到，但并不觉得比 OOM Killer 更为灵活，只不过是另一种 OOM Killer。

9.3 Low Memory Killer 驱动详解

在 Android 系统中，Low Memory Killer 驱动的实现文件是 drivers/misc/lowmemorykiller.c，将详细分析 Low Memory Killer 驱动的具体实现过程。

9.3.1 Low Memory Killer 驱动基础

在 Linux 中有一个名为 kswapd 的内核线程，当 Linux 回收存放分页时，线程 kswapd 会遍历一张 shrinker 链表并执行回调处理。在 Low Memory Killer 驱动系统中，会利用数组 lowmem_adj 和 lowmem_minfree 来作为评判当前内存不足的标准。假如当前系统中的空闲内存是 63MB 时，比 owmem_ minfree[3]小，那么就会选择在比 lowmem_adj[3]的 oom score adj 大的进程中找到一个 oom score adj 最大的将其杀掉。当两个进程的 oom score adj 一样时，会选择内存占用最多的杀掉。至于数组中的其他部分，则依次类推。

数组 lowmem_adj 的定义代码如下所示。

```
static int lowmem_adj[6] = {
	0,
	1,
	6,
	12,
};
static int lowmem_adj_size = 4;
```

数组 lowmem_minfre 的定义代码如下所示。

```
static size_t lowmem_minfree[6] = {
	3 * 512,		/* 6MB */
```

```
        2 * 1024,       /* 8MB */
        4 * 1024,       /* 16MB */
        16 * 1024,      /* 64MB */
};
static int lowmem_minfree_size = 4;
```

通过上述数组的实现代码可知，数组 lowmem_minfree 用于保存空闲内存的阈值，单位是一个页面 4KB。数组 lowmem_adj 用于保存每个阈值对应的优先级。

在 Low Memory Killer 机制中会首先进行初始化处理，此功能通过函数 lowmem_init()实现，具体实现代码如下所示。

```
static int __init lowmem_init(void)
{
        register_shrinker(&lowmem_shrinker);
        return 0;
}
```

在上述代码中，通过此接口注册之后，如果系统回收释放内存时，kswap 内核线程会遍历 Shrinker 链表，并调用通过此接口注册的 shrink()函数。另外，在函数 lowmem_in()中调用了函数 register_shrink()，功能是将 lowmem_shrink 加入 Shrinker List 中，当 kswapd 在遍历 Shrinker List 时调用此函数。

在退出 Low Memory Killer 时会调用函数 lowmem_ex()，具体实现代码如下所示。

```
static void __exit lowmem_exit(void)
{
    unregister_shrinker(&lowmem_shrinker);
}
```

在上述代码中，通过 unregister_shrinker 卸载了被注册的 lowmem_shrinker。

9.3.2　分析核心功能

Low Memory Killer 的核心功能是在函数 lowmem_shrink()中定义的，此函数的具体实现代码如下所示。

```
static int lowmem_shrink(int nr_to_scan, gfp_t gfp_mask)
{
        struct task_struct *p;
        struct task_struct *selected = NULL;
        int rem = 0;
        int tasksize;
        int i;
        int min_adj = OOM_ADJUST_MAX + 1;
        int selected_tasksize = 0;
        int selected_oom_adj;
        int array_size = ARRAY_SIZE(lowmem_adj);
        int other_free = global_page_state(NR_FREE_PAGES);
        int other_file = global_page_state(NR_FILE_PAGES);

        if (lowmem_adj_size < array_size)
                array_size = lowmem_adj_size;
        if (lowmem_minfree_size < array_size)
                array_size = lowmem_minfree_size;
        for (i = 0; i < array_size; i++) {
                if (other_free < lowmem_minfree[i] &&
                    other_file < lowmem_minfree[i]) {
```

```
				min_adj = lowmem_adj[i];
				break;
			}
		}
		if (nr_to_scan > 0)
			lowmem_print(3, "lowmem_shrink %d, %x, ofree %d %d, ma %d\n",
					nr_to_scan, gfp_mask, other_free, other_file,
					min_adj);
		rem = global_page_state(NR_ACTIVE_ANON) +
			global_page_state(NR_ACTIVE_FILE) +
			global_page_state(NR_INACTIVE_ANON) +
			global_page_state(NR_INACTIVE_FILE);
		if (nr_to_scan <= 0 || min_adj == OOM_ADJUST_MAX + 1) {
			lowmem_print(5, "lowmem_shrink %d, %x, return %d\n",
					nr_to_scan, gfp_mask, rem);
			return rem;
		}
		selected_oom_adj = min_adj;

		read_lock(&tasklist_lock);
		for_each_process(p) {
			struct mm_struct *mm;
			 int oom_adj;

			 task_lock(p);
			 mm = p->mm;
			 if (!mm) {
				 task_unlock(p);
				 continue;
			 }
			 oom_adj = mm->oom_adj;
			 if (oom_adj < min_adj) {
				 task_unlock(p);
				 continue;
			 }
			 tasksize = get_mm_rss(mm);
			 task_unlock(p);
			 if (tasksize <= 0)
				 continue;
			 if (selected) {
				 if (oom_adj < selected_oom_adj)
					 continue;
				 if (oom_adj == selected_oom_adj &&
					 tasksize <= selected_tasksize)
					 continue;
			 }
			 selected = p;
			 selected_tasksize = tasksize;
			 selected_oom_adj = oom_adj;
			 lowmem_print(2, "select %d (%s), adj %d, size %d, to kill\n",
					 p->pid, p->comm, oom_adj, tasksize);
```

```
	}
	if (selected) {
		lowmem_print(1, "send sigkill to %d (%s), adj %d, size %d\n",
				selected->pid, selected->comm,
				selected_oom_adj, selected_tasksize);
		force_sig(SIGKILL, selected);
		rem -= selected_tasksize;
	}
	lowmem_print(4, "lowmem_shrink %d, %x, return %d\n",
			nr_to_scan, gfp_mask, rem);
	read_unlock(&tasklist_lock);
	return rem;
}
```

在上述代码中，函数 lowmem_shrink()会首先计算当前空闲内存的大小，如果小于某个阈值，则以该阈值对应的优先级为基准，遍历各个进程，计算每个进程占用内存的大小，找出优先级大于基准优先级的进程，在这些进程中选择优先级最大的杀死，如果优先级相同，则选择占用内存最多的进程。

在函数 lowmem_shrink()的实现过程中，需要严格确定所定义数组 lowmem_adj 和数组 lowmem_minfree 的元素个数是否一致，如果不一致则以最小的为基准，这是因为需要通过比较 lowmem_ minfree 中的空闲存储空间的值以确定最小 min_adj 值。当满足一切条件时，首先通过其数组索引来寻找 lowmem_adj 中对应元素的值。然后检测 min_adj 的值是否是初始值 OOM_ADJUST_MAX+1，如果是则表示没有满足条件的 min_adj 值，否则进入下一步。然后使用循环对每一个进程块进行判断，通过 min_adj 来寻找满足条件的具体进程，这一过程主要包括对 oomkilladj 和 task_struct 判断的判断处理。最后对找到的进程进行 NULL，通过 force_sig(SIGKILL, selected)代码语句发送一条 SIGKILL 信号到内核，杀掉被选中的 selected 进程。

另外，在函数 lowmem_shrink()中还调用了 global_page_state()函数，此函数在文件/linux/include/linux/vmstat.h 中定义，具体实现代码如下所示。

```
static inline unsigned long global_page_state(enum zone_stat_item item)
{
	long x = atomic_long_read(&vm_stat[item]);
#ifdef CONFIG_SMP
	if (x < 0)
		x = 0;
#endif
	return x;
}
```

在上述代码中，参数 zone_stat_item 是一个枚举，此枚举在文件 linux/mmzone.h 中定义，具体实现代码如下所示。

```
enum zone_stat_item {
    NR_FREE_PAGES,
    NR_LRU_BASE,
    NR_INACTIVE_ANON = NR_LRU_BASE,
    NR_ACTIVE_ANON,
    NR_INACTIVE_FILE,
    NR_ACTIVE_FILE,
#ifdef CONFIG_UNEVICTABLE_LRU
    NR_UNEVICTABLE,
    NR_MLOCK,
#else
    NR_UNEVICTABLE = NR_ACTIVE_FILE,        /*避免编译错误*/
```

```
    NR_MLOCK = NR_ACTIVE_FILE,
#endif
    NR_ANON_PAGES,                /*匿名映射页面*/
    NR_FILE_MAPPED,               /*映射页面*/
    NR_FILE_PAGES,
    NR_FILE_DIRTY,
    NR_WRITEBACK,
    NR_SLAB_RECLAIMABLE,
    NR_SLAB_UNRECLAIMABLE,
    NR_PAGETABLE,
    NR_UNSTABLE_NFS,
    NR_BOUNCE,
    NR_VMSCAN_WRITE,
    NR_WRITEBACK_TEMP,            /*使用临时缓冲区*/
#ifdef CONFIG_NUMA
    NUMA_HIT,                     /*在预定节点上分配*/
    NUMA_MISS,                    /*在非预定节点上分配*/
    NUMA_FOREIGN,
    NUMA_INTERLEAVE_HIT,
    NUMA_LOCAL,                   /*从本地页面分配*/
    NUMA_OTHER,                   /*从其他节点分配 */
#endif
    NR_VM_ZONE_STAT_ITEMS };
```

在核心函数 lowmem_shrink()中经过 for_each 遍历处理，selected 就是我们选出要释放掉的 bad 进程，此进程具有如下两个条件。

☑ oom_adj 大于当前警戒阈值并且最大。

☑ 在同样大小的 oom_adj 中占用最多的内存。

9.3.3 设置用户接口

在杀掉指定的进程后，函数 lowmem_shrink()最后会释放掉这个进程的内存，通过 force_sig(SIGKILL, selected)向进程发送一个不可以忽略或阻塞的 SIGKILL 信号。在启动 Android 系统时会读取配置文件/init.rc，在此文件中定义了相应的属性供 AP 使用，并设置了上述接口参数，具体代码如下所示。

```
#killed by the kernel.These are used in
ActivityManagerService.
setprop ro.FOREGROUND_APP_ADJ 0
setprop ro.VISIBLE_APP_ADJ 1
setprop ro.SECONDARY_SERVER_ADJ 2
setprop ro.BACKUP_APP_ADJ 2
setprop ro.HOME_APP_ADJ 4
setprop ro.HIDDEN_APP_MIN_ADJ 7
setprop ro.CONTENT_PROVIDER_ADJ 14
setprop ro.EMPTY_APP_ADJ 15
#Define the memory thresholds at which the above process
classes will
#be killed.These numbers are in pages(4k).
setprop ro.FOREGROUND_APP_MEM 1536
setprop ro.VISIBLE_APP_MEM 2048
setprop ro.SECONDARY_SERVER_MEM 4096
```

```
setprop ro.BACKUP_APP_MEM 4096
setprop ro.HOME_APP_MEM 4096
setprop ro.HIDDEN_APP_MEM 5120
setprop ro.CONTENT_PROVIDER_MEM 5632
setprop ro.EMPTY_APP_MEM 6144
#Write value must be consistent with the above properties.
#Note that the driver only supports 6 slots,so we have HOME_APP
at the
#same memory level as services.
write/sys/module/lowmemorykiller/parameters/adj
0,1,2,7,14,15
write/sys/module/lowmemorykiller/parameters/minfree
1536,2048,4096,5120,5632,6144
#Set init its forked children's oom_adj.
write/proc/1/oom_adj-16
```

我们截取上述文件 init.rc 中的相关配置代码片段，具体代码如下所示。

```
# Write value must be consistent with the above properties.
    write /sys/module/lowmemorykiller/parameters/adj 0,1,2,7,14,15
    write /proc/sys/vm/overcommit_memory 1
    write /sys/module/lowmemorykiller/parameters/minfree 1536,2048,4096,5120,5632,6144
    class_start default
```

我们可以通过如下值来配置阈值表。

- ☑ /sys/module/lowmemorykiller/parameters/adj
- ☑ /sys/module/lowmemorykiller/parameters/minfree

同样的道理，通过 write/proc/<PID>/oom_adj 可以设置进程 oom_adj 不被杀死。具体方法是在文件 init.rc 中，将 init 进程的 pid 配置为 1，将 omm_adj 配置为-16，具体代码如下所示。

```
# Set init its forked children's oom_adj.
write /proc/1/oom_adj -16
```

到此为止，Android 系统中的 Low Memory Killer 驱动基本原理介绍完毕。由此可见，进程 omm_adj 的大小和进程的类型以及进程被调度的次序有关，可以在 ActivityManagerService 中清楚地看到进程的具体类型，具体代码如下所示。

```
static final int EMPTY_APP_ADJ;
static final int HIDDEN_APP_MAX_ADJ;
static final int HIDDEN_APP_MIN_ADJ;
static final int HOME_APP_ADJ;
static final int BACKUP_APP_ADJ;
static final int SECONDARY_SERVER_ADJ;
static final int HEAVY_WEIGHT_APP_ADJ;
static final int PERCEPTIBLE_APP_ADJ;
static final int VISIBLE_APP_ADJ;
static final int FOREGROUND_APP_ADJ;
static final int CORE_SERVER_ADJ = -12;
static final int SYSTEM_ADJ = -16;
```

在 ActivityManagerService 中定义了各种进程的 oom_adj，其中 CORE_SERVER_ADJ 代表一些核心服务的 omm_adj，数值为-12，参数值为 12 的进程永远也不会被杀死。

至于其他的未赋值，也都在 static 块中进行了初始化处理，具体是通过文件 system/rootdir/init.rc 进行配置的。在文件 init.rc 中的相关代码如下所示。

```
# Define the oom_adj values for the classes of processes that can be
# killed by the kernel.   These are used in ActivityManagerService.
    setprop ro.FOREGROUND_APP_ADJ 0
    setprop ro.VISIBLE_APP_ADJ 1
    setprop ro.SECONDARY_SERVER_ADJ 2
    setprop ro.HIDDEN_APP_MIN_ADJ 7
    setprop ro.CONTENT_PROVIDER_ADJ 14
    setprop ro.EMPTY_APP_ADJ 15
# Define the memory thresholds at which the above process classes will
# be killed.   These numbers are in pages (4k).
    setprop ro.FOREGROUND_APP_MEM 1536
    setprop ro.VISIBLE_APP_MEM 2048
    setprop ro.SECONDARY_SERVER_MEM 4096
    setprop ro.HIDDEN_APP_MEM 5120
    setprop ro.CONTENT_PROVIDER_MEM 5632
    setprop ro.EMPTY_APP_MEM 6144
```

通过上述代码可知最容易被杀死的是 EMPTY_APP，最难被杀死的进程是 CONTENT_PROVIDER 和 FOREGROUND 的进程。

9.4 实战演练——从内存池获取对象

经过分析 Android 系统内置的内存驱动架构后，本节将详细讲解在 Linux 底层内核开发自己的驱动程序的方法。笔者编写的驱动程序文件是 neicun_cache.c，具体实现代码如下所示。

```
//头文件
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/slab.h>
#include <linux/mempool.h>
static mempool_t *neicun;
struct data
{
	char *name;
	int value;
//nr 表示当获得对象时在 elements 中的索引
	int nr;
}*datap;
void *mempool_alloc_fun(gfp_t gfp_mask, void *pool_data)
{
//如果 neicun 为空，则需要创建新对象
	if (!neicun)
	{
//为结构体 data 分配内存空间
		pool_data = kmalloc(sizeof(struct data), GFP_KERNEL);
		printk("alloc struct data\n");
	}
//通过 mempool_alloc()函数调用函数 alloc()
```

```
	else
	{
		if (neicun->curr_nr < neicun->min_nr)
		{
//从数组 element 中获取一个 element
			void *element = neicun->elements[neicun->curr_nr];
			if (element)
			{
//用当前的 curr_nr 设置 data_nr
				((struct data*) element)->nr = neicun->curr_nr;
				neicun->elements[neicun->curr_nr--] = NULL;
			}
			return element;
		}
		else
			return NULL;
	}
	return pool_data;
}
//下面是内存池的 free()函数
void mempool_free_fun(void *element, void *pool_data)
{
	if (element)
	{
		kfree(element);
		printk("free struct data\n");
	}
}
//Linux 驱动的 init()函数
static int __init demo_init(void)
{
//创建可以有 5 个对象的内存池
	neicun = mempool_create(5, mempool_alloc_fun, mempool_free_fun, NULL);
	neicun->curr_nr = neicun->curr_nr - 1;
	datap = mempool_alloc(neicun, GFP_KERNEL);
	datap->name = "mempool data";
	datap->value = 4321;
	printk(KERN_ALERT "demo_init.\n");
	return 0;
}
//Linux 驱动的 exit()函数
static void __exit demo_exit(void)
{
	if (datap)
	{
//输出 data.name
		printk("data.name=%s\n", datap->name);
//输出 data.value
		printk("data.value=%d\n", datap->value);
		if (neicun && datap)
		{
```

```
                neicun->curr_nr = datap->nr;
                mempool_free(datap, neicun);
//将 curr_nr 设置为 min_nr
                neicun->curr_nr = neicun->min_nr;
//销毁内存池
                mempool_destroy(neicun);
            }
        }
        printk(KERN_ALERT "demo_exit.\n");
}
MODULE_LICENSE("GPL");
module_init(demo_init);
module_exit(demo_exit);
```

接下来可以使用脚本文件 build.sh 来编译和安装上述 neicun_cache 驱动，并在 Linux 端执行 dmesg 命令。执行后会成功创建和销毁内存池，在销毁内存池时会从内存池中分配结构体 data。

9.5　实战演练——使用用户程序读取内核空间的数据

在本节的驱动程序实例中，会将 Buffer 指向的内存空间映射到用户空间，并在驱动中向 Buffer 写入一个字符串，最后在用户空间程序中读取这个字符串的值。本实例的驱动程序文件是 mmap_ gongxiang.c，具体实现代码如下所示。

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>

//定义设备文件名
#define DEVICE_NAME "mmap_shared"
#define BUFFER_SIZE 4096
static char *buffer;

//指向映射内存的指针
static void demo_vma_open(struct vm_area_struct *vma)
{
        printk(KERN_INFO"VMA open.\n");
}
static void demo_vma_close(struct vm_area_struct *vma)
{
        printk(KERN_INFO"VMA close.\n");
}
static struct vm_operations_struct remap_vm_ops =
{ .open = demo_vma_open, .close = demo_vma_close};
static int demo_mmap(struct file *filp, struct vm_area_struct *vma)
{
```

```
    unsigned long physics = virt_to_phys((void*) (unsigned long) buffer); //((unsigned long )buffer)-PAGE_
OFFSET;
    unsigned long mypfn = physics >> PAGE_SHIFT;
    unsigned long vmsize = vma->vm_end - vma->vm_start;
    printk(KERN_INFO"demo_mmap called\n");
    if (vmsize > BUFFER_SIZE)
        return -EINVAL;
    vma->vm_ops = &remap_vm_ops;
    vma->vm_flags |= VM_RESERVED;
    demo_vma_open(vma);
    if (remap_pfn_range(vma, vma->vm_start, mypfn, vmsize, vma->vm_page_prot))
        return -EAGAIN;
    return 0;
}
static struct file_operations dev_fops =
{ .owner = THIS_MODULE, .mmap = demo_mmap };
//描述设备文件信息
static struct miscdevice misc =
{ .minor = MISC_DYNAMIC_MINOR, .name = DEVICE_NAME, .fops = &dev_fops };
static int __init demo_init(void)
{
    int ret;
    struct page *page;
    //建立设备文件
    ret = misc_register(&misc);
    buffer = kmalloc(BUFFER_SIZE, GFP_KERNEL);
    for (page = virt_to_page(buffer); page < virt_to_page(buffer + BUFFER_SIZE);page++)
    {
        //设置当前页为保留状态
        SetPageReserved(page);
    }memset(buffer, 0, BUFFER_SIZE);
    strcpy(buffer, "mmap_shared_success!!!!!!!\n");
    printk(KERN_INFO "demo_init.\n");
    return ret;
}
static void __exit demo_exit(void)
{
    struct page *page;
    //删除设备文件
    misc_deregister(&misc);
    for (page = virt_to_page(buffer); page < virt_to_page(buffer + BUFFER_SIZE);page++)
    {
        //删除页的保留状态
        ClearPageReserved(page);
    }
    printk(KERN_INFO "demo_exit.\n");
}
MODULE_LICENSE("GPL");
module_init(demo_init);
module_exit(demo_exit);
```

再看读取内存映射文件 user_mmap.c，具体实现代码如下所示。

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/mman.h>
#include <stdio.h>
#define PAGE_SIZE (4*1024)
int main()
{
	int fd;
	void *start;
	fd = open("/dev/mmap_shared", O_RDWR);
	start = mmap(NULL, PAGE_SIZE, PROT_READ, MAP_PRIVATE, fd, 0);
	if (start == MAP_FAILED)
	{
		printf("mmap error\n");
		return 0;
	}
	puts(start);
	munmap(start, PAGE_SIZE);
	close(fd);
}
```

接下来就可以使用脚本文件 build.sh 来编译和安装上述驱动了，在 Linux 端执行 user_mmap 命令后会输出“mmap_shared_success!!!!!!!!”提示信息。

第 3 篇



典型驱动移植篇

第 10 章　电源管理驱动

Android 系统中的电源管理系统是 Android Power Management，这是一个基于标准 Linux 电源管理的轻量级 Android 电源管理系统。整个电源管理系统分为 4 大部分，分别是应用层、框架层、HAL 层和 Kernel 层。本章将详细讲解 Android 系统中 Power Management 系统驱动的基本架构知识，为读者学习本书后面的知识打下基础。

10.1　Power Management 架构基础

在 Android 系统中，电源管理模块 Android Power Management 的整体架构如图 10-1 所示。

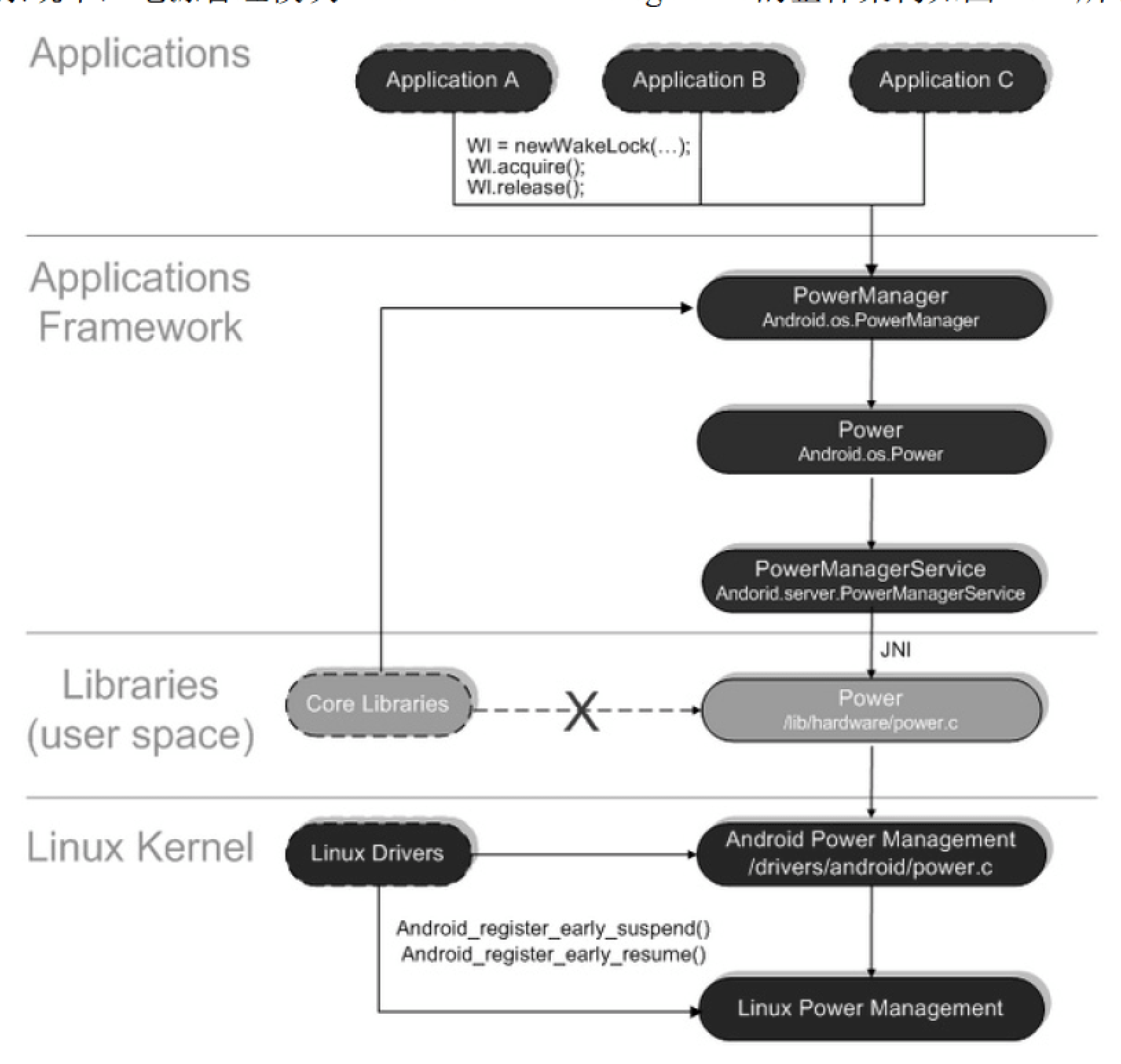


图 10-1　Android Power Management（电源管理系统）的整体架构

从图 10-1 所示的架构可以看出，电源管理主要是通过锁和定时器来切换系统的状态，使系统的功耗降至最低。为了使读者了解的更加清晰，请再看图 10-2 所示的电源管理详细架构图。

由此可见，整个电源管理模块分为 4 大部分，分别是应用层、框架层、HAL 层和 Kernel 层。整个运作流程设计如下。

wake_lock->setScreenState(off)->request_suspend_state->early_suspend->wake_unlock->suspend->late suspend->sleep->wakeup->early resume->resume->late resume

本章将详细分析上述 4 个层次的具体实现过程。

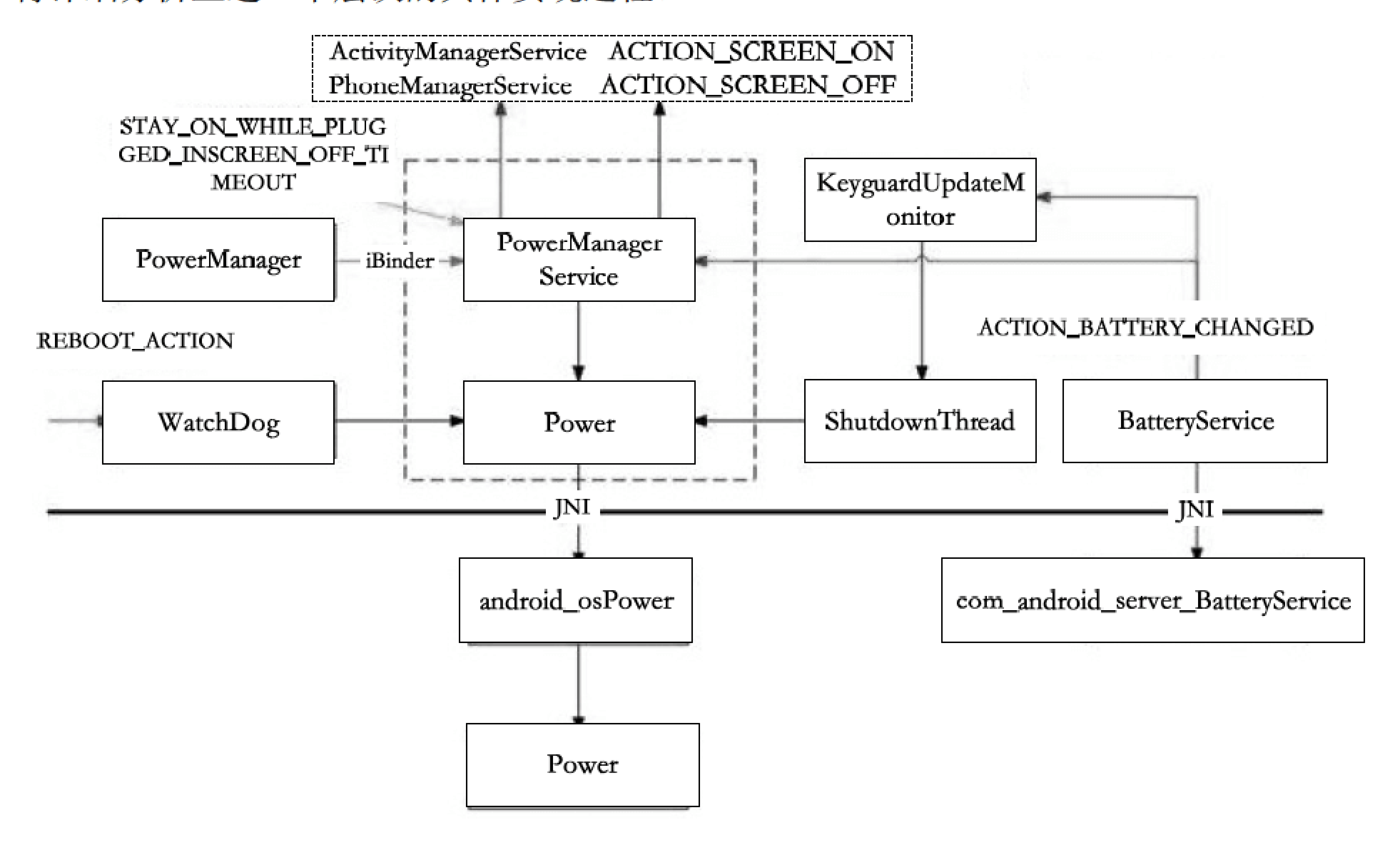


图 10-2　电源管理系统的详细架构图

10.2　分析 Framework 层

我们知道，Android 系统的 Framework 层是接口层，为上面的应用层提供了拿来即用的接口。在 Android Power Management 系统中，Framework 层涉及如下的文件。

☑　frameworks/base/core/java/android/os/PowerManager.java

☑　frameworks/base/services/java/com/android/server/power/PowerManagerService.java

本节将详细介绍 Power Management 系统中 Framework 层文件的实现过程。

10.2.1　文件 PowerManager.java

文件 PowerManager.java 是提供给应用层调用的，最终的核心是在文件 PowerManagerService.java 中实现。在此文件 PowerManager.java 中定义了类 android.os.PowerManager，功能是控制设备的电源状态切换。获取 PowerManager 在 getSystemService(Context.POWER_SERVICE)获取对象时是通过构造函数 PowerManager (IPowerManagerservice, Handler handler){}来创建的，而此处 IPowerManager 则是创建 PowerManager 实例的核心，而 IPowerManager 则是由 PowerManagerService 实现的，所以从本质上说，PowerManager 的大部分方法是由 PowerManagerService 实现的。

下面将详细分析文件 PowerManager.java 的具体实现流程。

（1）定义类 PowerManager 和接口函数，具体代码如下所示。

```
public final class PowerManager {
    private static final String TAG = "PowerManager";
```

```
    public static final int PARTIAL_WAKE_LOCK = 0x00000001;
    @Deprecated
    public static final int SCREEN_DIM_WAKE_LOCK = 0x00000006;
    @Deprecated
    public static final int SCREEN_BRIGHT_WAKE_LOCK = 0x0000000a;
    @Deprecated
    public static final int FULL_WAKE_LOCK = 0x0000001a;
    public static final int PROXIMITY_SCREEN_OFF_WAKE_LOCK = 0x00000020;
    public static final int WAKE_LOCK_LEVEL_MASK = 0x0000ffff;
    public static final int ACQUIRE_CAUSES_WAKEUP = 0x10000000;
    public static final int ON_AFTER_RELEASE = 0x20000000;
    public static final int WAIT_FOR_PROXIMITY_NEGATIVE = 1;
    public static final int BRIGHTNESS_ON = 255;
```

（2）定义对外接口函数，以实现对电源状态的控制管理。其中函数 goToSleep()的功能是强制设备进入 Sleep 状态，函数 wakeUp()的功能是强制设备进入 wakeUp 状态。

```
public void goToSleep(long time) {
    try {
        mService.goToSleep(time, GO_TO_SLEEP_REASON_USER);
    } catch (RemoteException e) {
    }
}
public void wakeUp(long time) {
    try {
        mService.wakeUp(time);
    } catch (RemoteException e) {
    }
}
```

（3）定义函数 userActivity()，功能是当发生 User Activity 事件时，电源设备会被切换到 Full on 的状态，并同时 Reset Screen off timer，具体代码如下所示。

```
public void userActivity(long when, boolean noChangeLights) {
    try {
        mService.userActivity(when, USER_ACTIVITY_EVENT_OTHER,
                noChangeLights ? USER_ACTIVITY_FLAG_NO_CHANGE_LIGHTS : 0);
    } catch (RemoteException e) {
    }
}
```

10.2.2 提供 PowerManager 功能

文件 PowerManagerService.java 是 Power Management 系统中整个 Framework 层文件的核心，这个类的作用就是提供 PowerManager 的功能，以及整个电源管理状态机的运行。PowerManagerService 服务是 Android 系统的上层的电源管理服务，主要负责系统待机、屏幕背光、按键背光、键盘背光以及用户事件的处理。通过锁的申请与释放以及默认的待机时间来控制系统的待机状态；通过系统默认关闭屏的时间以及用户操作的事件状态控制背光的亮和暗。另外，PowerManagerService 服务还包括了光线、距离传感器上层查询与控制，LCD 亮度的调节最终也是由该服务完成的。在本章前面介绍的文件 PowerManager.java 只是定义了各个对外接口函数，而文件 PowerManagerService.java 则定义了各个接口函数的具体实现。

下面将详细分析文件 PowerManagerService.java 的具体实现过程。

1．定义常量和变量

在文件的开始定义服务类 PowerManagerService，并定义需要的变量和常量，主要实现代码如下所示。

```
private static final int LOCK_MASK = PowerManager.PARTIAL_WAKE_LOCK
                                    | PowerManager.SCREEN_DIM_WAKE_LOCK
                                    | PowerManager.SCREEN_BRIGHT_WAKE_LOCK
                                    | PowerManager.FULL_WAKE_LOCK
                                    | PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK;

//                        time since last state:               time since last event:
// The short keylight delay comes from secure settings; this is the default.
private static final int SHORT_KEYLIGHT_DELAY_DEFAULT = 6000; // t+6 sec
private static final int MEDIUM_KEYLIGHT_DELAY = 15000;          // t+15 sec
private static final int LONG_KEYLIGHT_DELAY = 6000;             // t+6 sec
private static final int LONG_DIM_TIME = 7000;                   // t+N-5 sec

// How long to wait to debounce light sensor changes in milliseconds
private static final int LIGHT_SENSOR_DELAY = 2000;                      //光线传感器时延

// light sensor events rate in microseconds
private static final int LIGHT_SENSOR_RATE = 1000000;                    //光线传感器频率

// For debouncing the proximity sensor in milliseconds
private static final int PROXIMITY_SENSOR_DELAY = 1000;                  //距离传感器时延

// trigger proximity if distance is less than 5 cm
private static final float PROXIMITY_THRESHOLD = 5.0f;                   //距离传感器距离范围

// Cached secure settings; see updateSettingsValues()
private int mShortKeylightDelay = SHORT_KEYLIGHT_DELAY_DEFAULT;    //键盘灯短暂时延

// Default timeout for screen off, if not found in settings database = 15 seconds.
private static final int DEFAULT_SCREEN_OFF_TIMEOUT = 15000;//默认屏幕超时时间，从 Settings 中获取

// flags for setPowerState
private static final int SCREEN_ON_BIT          = 0x00000001;
private static final int SCREEN_BRIGHT_BIT      = 0x00000002;
private static final int BUTTON_BRIGHT_BIT      = 0x00000004;
private static final int KEYBOARD_BRIGHT_BIT    = 0x00000008;
private static final int BATTERY_LOW_BIT        = 0x00000010;

// values for setPowerState

private static final int SCREEN_OFF             = 0x00000000;          //屏幕灭掉，进入睡眠状态

private static final int SCREEN_DIM             = SCREEN_ON_BIT;       //屏幕灭掉，依然在工作状态

private static final int SCREEN_BRIGHT          = SCREEN_ON_BIT | SCREEN_BRIGHT_BIT;//屏幕亮，处于工
作状态

private static final int SCREEN_BUTTON_BRIGHT   = SCREEN_BRIGHT | BUTTON_BRIGHT_BIT;//屏幕亮，
```

```
按键灯亮

// SCREEN_BUTTON_BRIGHT == screen on, screen, button and keyboard backlights bright
private static final int ALL_BRIGHT             = SCREEN_BUTTON_BRIGHT | KEYBOARD_BRIGHT_BIT;//按
键灯亮，键盘灯亮

// used for noChangeLights in setPowerState()
private static final int LIGHTS_MASK            = SCREEN_BRIGHT_BIT | BUTTON_BRIGHT_BIT |
KEYBOARD_BRIGHT_BIT;                                        //屏幕亮，按键灯亮，键盘灯亮

boolean mAnimateScreenLights = true;

static final int ANIM_STEPS = 60/4;
// Slower animation for autobrightness changes
static final int AUTOBRIGHTNESS_ANIM_STEPS = 60;

// These magic numbers are the initial state of the LEDs at boot.   Ideally
// we should read them from the driver, but our current hardware returns 0
// for the initial value.   Oops!
static final int INITIAL_SCREEN_BRIGHTNESS = 255;                          //屏幕初始状态亮
static final int INITIAL_BUTTON_BRIGHTNESS = Power.BRIGHTNESS_OFF;         //按键灯初始状态灭
static final int INITIAL_KEYBOARD_BRIGHTNESS = Power.BRIGHTNESS_OFF;   //键盘灯初始状态灭

private final int MY_UID;
private final int MY_PID;

private boolean mDoneBooting = false;
private boolean mBootCompleted = false;                                    //开机完成标志位
private int mStayOnConditions = 0;
private final int[] mBroadcastQueue = new int[] { -1, -1, -1 };
private final int[] mBroadcastWhy = new int[3];
private boolean mPreparingForScreenOn = false;
private boolean mSkippedScreenOn = false;
private boolean mInitialized = false;
private int mPartialCount = 0;
private int mPowerState;
// mScreenOffReason can be WindowManagerPolicy.OFF_BECAUSE_OF_USER,
// WindowManagerPolicy.OFF_BECAUSE_OF_TIMEOUT or WindowManagerPolicy.OFF_BECAUSE_OF_PROX_
SENSOR
private int mScreenOffReason;
private int mUserState;
private boolean mKeyboardVisible = false;
private int mStartKeyThreshold = 0;
private boolean mUserActivityAllowed = true;
private int mProximityWakeLockCount = 0;
private boolean mProximitySensorEnabled = false;              //距离传感器是否可用
private boolean mProximitySensorActive = false;               //当前距离传感器是否工作
private int mProximityPendingValue = -1; // -1 == nothing, 0 == inactive, 1 == active
private long mLastProximityEventTime;
private int mScreenOffTimeoutSetting;                         //屏幕超时设置
private int mMaximumScreenOffTimeout = Integer.MAX_VALUE;
```

```
private int mKeylightDelay;
private int mDimDelay;
private int mScreenOffDelay;
private int mWakeLockState;
private long mLastEventTime = 0;
private long mScreenOffTime;
private volatile WindowManagerPolicy mPolicy;
private final LockList mLocks = new LockList();
private Intent mScreenOffIntent;
private Intent mScreenOnIntent;
private LightsService mLightsService;                    //系统 LightsService
private Context mContext;
private LightsService.Light mLcdLight;                   //屏
private LightsService.Light mButtonLight;                //按键灯
private LightsService.Light mKeyboardLight;              //键盘灯（若有实体输入法按键）
private LightsService.Light mAttentionLight;             //通知等（若有信号灯）
private UnsynchronizedWakeLock mBroadcastWakeLock;       //广播同步锁
private UnsynchronizedWakeLock mStayOnWhilePluggedInScreenDimLock;
private UnsynchronizedWakeLock mStayOnWhilePluggedInPartialLock;
private UnsynchronizedWakeLock mPreventScreenOnPartialLock;
private UnsynchronizedWakeLock mProximityPartialLock;
private HandlerThread mHandlerThread;
private HandlerThread mScreenOffThread;
private Handler mScreenOffHandler;
private Handler mHandler;

//计时器线程，主要完成管理屏幕超时操作，当有用户点击屏幕时，该计时器重新开始计时，直到无任何操作，且
到屏幕延时最大时间，将屏幕灭掉
private final TimeoutTask mTimeoutTask = new TimeoutTask();
private final BrightnessState mScreenBrightness
= new BrightnessState(SCREEN_BRIGHT_BIT);                //亮度管理
private boolean mStillNeedSleepNotification;
private boolean mIsPowered = false;
private IActivityManager mActivityService;
private IBatteryStats mBatteryStats;
private BatteryService mBatteryService;                  //电池服务
private SensorManager mSensorManager;                    //Sensor 管理器
private Sensor mProximitySensor;                         //距离传感器
private Sensor mLightSensor;                             //光线传感器
private Sensor mLightSensorKB;                           //光线传感器
private boolean mLightSensorEnabled;                     //光线传感器是否可用
private float mLightSensorValue = -1;
private boolean mProxIgnoredBecauseScreenTurnedOff = false;
private int mHighestLightSensorValue = -1;
private boolean mLightSensorPendingDecrease = false;
private boolean mLightSensorPendingIncrease = false;
private float mLightSensorPendingValue = -1;
private int mLightSensorScreenBrightness = -1;
private int mLightSensorButtonBrightness = -1;
private int mLightSensorKeyboardBrightness = -1;
private boolean mDimScreen = true;
```

```
private boolean mIsDocked = false;
private long mNextTimeout;
private volatile int mPokey = 0;
private volatile boolean mPokeAwakeOnSet = false;
private volatile boolean mInitComplete = false;
private final HashMap<IBinder,PokeLock> mPokeLocks = new HashMap<IBinder,PokeLock>();
// mLastScreenOnTime is the time the screen was last turned on
private long mLastScreenOnTime;
private boolean mPreventScreenOn;
private int mScreenBrightnessOverride = -1;
private int mButtonBrightnessOverride = -1;
private int mScreenBrightnessDim;
private boolean mUseSoftwareAutoBrightness;
private boolean mAutoBrightessEnabled;
private int[] mAutoBrightnessLevels;
private int[] mLcdBacklightValues;
private int[] mButtonBacklightValues;
private int[] mKeyboardBacklightValues;
private int mLightSensorWarmupTime;
boolean mUnplugTurnsOnScreen;
private int mWarningSpewThrottleCount;
private long mWarningSpewThrottleTime;
private int mAnimationSetting = ANIM_SETTING_OFF;

// Must match with the ISurfaceComposer constants in C++.
private static final int ANIM_SETTING_ON = 0x01;
private static final int ANIM_SETTING_OFF = 0x10;

// Used when logging number and duration of touch-down cycles
private long mTotalTouchDownTime;
private long mLastTouchDown;
private int mTouchCycles;

// could be either static or controllable at runtime
private static final boolean mSpew = false;
private static final boolean mDebugProximitySensor = (false || mSpew);
private static final boolean mDebugLightSensor = (false || mSpew);

private native void nativeInit();
private native void nativeSetPowerState(boolean screenOn, boolean screenBright);
private native void nativeStartSurfaceFlingerAnimation(int mode);
```

在文件 PowerManagerService 中，其中需要重点说明的变量如下。

☑ mDirty：功能是表示 power state 的变化，在系统中一共定义了 12 个与之类似的变化，每一个 state 对应一个固定的数字，都是 2 的倍数。这样当有若干个状态一起变化时就会按位取或，这样不但会得到一个唯一的结果，而且可以准确地标示出各个状态的变化。

☑ mWakefulness：功能是标示 device 处于是醒着的还是睡眠中的状态，或者处于两者之间的一种状态。这个状态和 display 的电源状态不同，display 的电源状态是独立管理的。这个变量用来表示 DIRTY_WAKEFULNESS 这个 power state 下的一个具体的内容。例如当系统进入 Dreaming 时，首先变化的是 mDirty，在 mDirty 中对 DIRTY_WAKEFULNESS 位置位，这说明系统中的 DIRTY_

WAKEFULNESS 发生了变化。此时只是知道 DIRTY_WAKEFULNESS 发生了变化，并不知道 wakefulness 发生了怎样的变化。如果需要进一步了解系统 wakefulness 变成了什么，则需要查看 mWakefulness 的内容。

2．开机启动及处理

（1）当 Android 系统启动时，会调用文件 SystemServer.java 中的接口函数 run()，在此函数中将 power 服务加入到系统服务中，具体代码如下所示。

```
power = new PowerManagerService();
ServiceManager.addService(Context.POWER_SERVICE, power);
```

其实在文件 SystemServer.java 中还定义了多种类型的服务加入到了系统服务中，具体代码如下所示。

```
try {
    // Wait for installd to finished starting up so that it has a chance to
    // create critical directories such as /data/user with the appropriate
    // permissions. We need this to complete before we initialize other services.
    Slog.i(TAG, "Waiting for installd to be ready.");
    installer = new Installer();
    installer.ping();
    Slog.i(TAG, "Power Manager");
    power = new PowerManagerService();
    ServiceManager.addService(Context.POWER_SERVICE, power);
    Slog.i(TAG, "Activity Manager");
    context = ActivityManagerService.main(factoryTest);
    Slog.i(TAG, "Display Manager");
    display = new DisplayManagerService(context, wmHandler, uiHandler);
    ServiceManager.addService(Context.DISPLAY_SERVICE, display, true);
    Slog.i(TAG, "Telephony Registry");
    telephonyRegistry = new TelephonyRegistry(context);
    ServiceManager.addService("telephony.registry", telephonyRegistry);
    Slog.i(TAG, "Scheduling Policy");
    ServiceManager.addService(Context.SCHEDULING_POLICY_SERVICE,
            new SchedulingPolicyService());
    AttributeCache.init(context);
    if (!display.waitForDefaultDisplay()) {
            reportWtf("Timeout waiting for default display to be initialized.",
                    new Throwable());
    }
    Slog.i(TAG, "Package Manager");
    // Only run "core" apps if we're encrypting the device.
    String cryptState = SystemProperties.get("vold.decrypt");
    if (ENCRYPTING_STATE.equals(cryptState)) {
        Slog.w(TAG, "Detected encryption in progress - only parsing core apps");
        onlyCore = true;
    } else if (ENCRYPTED_STATE.equals(cryptState)) {
        Slog.w(TAG, "Device encrypted - only parsing core apps");
        onlyCore = true;
    }
    pm = PackageManagerService.main(context, installer,
        factoryTest != SystemServer.FACTORY_TEST_OFF,
        onlyCore);
```

```
boolean firstBoot = false;
try {
    firstBoot = pm.isFirstBoot();
} catch (RemoteException e) {
}
ActivityManagerService.setSystemProcess();
Slog.i(TAG, "Entropy Mixer");
ServiceManager.addService("entropy", new EntropyMixer(context));
Slog.i(TAG, "User Service");
ServiceManager.addService(Context.USER_SERVICE,
        UserManagerService.getInstance());
mContentResolver = context.getContentResolver();
// The AccountManager must come before the ContentService
try {
    Slog.i(TAG, "Account Manager");
    accountManager = new AccountManagerService(context);
    ServiceManager.addService(Context.ACCOUNT_SERVICE, accountManager);
} catch (Throwable e) {
    Slog.e(TAG, "Failure starting Account Manager", e);
}
Slog.i(TAG, "Content Manager");
contentService = ContentService.main(context,
        factoryTest == SystemServer.FACTORY_TEST_LOW_LEVEL);
Slog.i(TAG, "System Content Providers");
ActivityManagerService.installSystemProviders();
Slog.i(TAG, "Lights Service");
lights = new LightsService(context);
Slog.i(TAG, "Battery Service");
battery = new BatteryService(context, lights);
ServiceManager.addService("battery", battery);
Slog.i(TAG, "Vibrator Service");
vibrator = new VibratorService(context);
ServiceManager.addService("vibrator", vibrator);
```

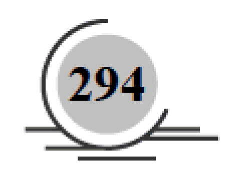
（2）当光感服务与电池管理服务都开始启动后，开始进行初始化 power 服务的工作。初始化工作是通过调用函数 init()实现的，具体实现用如下加粗代码所示。

```
// only initialize the power service after we have started the
// lights service, content providers and the battery service.
power.init(context, lights, ActivityManagerService.self(), battery,
    BatteryStatsService.getService(), display);
Slog.i(TAG, "Alarm Manager");
alarm = new AlarmManagerService(context);
ServiceManager.addService(Context.ALARM_SERVICE, alarm);
Slog.i(TAG, "Init Watchdog");
Watchdog.getInstance().init(context, battery, power, alarm,
    ActivityManagerService.self());
```

在函数 init()中实现了一些基本的初始化工作，包括将 lights 和 battery 两个服务实例传入到 power 服务中，这两个服务将与 power 进行交互。除此之外，还开启了如下两个线程。

☑ 开启处理亮度动画线程函数 mScreenBrightnessAnimator.start()。

mScreenBrightnessAnimator 是 PowerManagerService 子类 ScreenBrightnessAnimator 的实例，演示代码如



下所示。

```
mHandlerThread = new HandlerThread("PowerManagerService")
mHandlerThread.start( );
```

☑　初始化线程 initThread。

当使用 start()函数启动电源管理服务后会调用 run 接口，并在其中回调到子类中的 protected void onLooperPrepared()中，该接口又调用到 initInThread()，功能是实现一些值的初始化工作，并标识为“mInitComplete = true;”，发现为 true 的标识后 mHandlerThread.notifyAll()会通知创建 mHandlerThreadLooper 实例，该实例在函数 systemReady()中被 SystemSensorManager(mHandlerThread.getLooper())所使用。另外，在 initThread 线程中还实现了对 mSettings.addObserver(settingsObserver)的监听，如果用户在系统中变更了关于背光时间或是否启用光感等服务，PowerManagerService 可以获取到最新的状态值，这一功能需要使用函数 update()进行更新。

（3）继续看函数 init()，最后调用函数 forceUserActivityLocked()关闭服务，并标志初始化工作完成。

3．与系统其他模块之间的交互

在 Android 系统中，PowerManagerService 作为 Framework 中重要的能源管理模块，除了与应用程序交互之外，还需要与系统中其他模块进行配合，在提供良好的能源管理的同时提供友好的用户体验。Android 系统除了提供公共接口与其他模块交互外，还提供了 BroadCast 机制以对系统中发生的重要变化做出反应。在表 10-1 中列出了在 PowerManagerService 中注册的 Receiver，以及这些 Receiver 监听的事件和处理方法。

表 10-1　PowerManagerService 中注册的 Receiver 说明

BatteryReceiver	ACTION_BATTERY_CHANGED	handleBatterStateChangeLocked()
BootCompleteReceiver	ACTION_BOOT_COMPLETED	startWatchingForBootAnimationFinished()
userSwitchReceiver	ACTION_USER_SWITCHED	handleSettingsChangedLocked
DockReceiver	ACTION_DOCK_EVENT	updatePowerStateLocked
DreamReceiver	ACTION_DREAMING_STARTED ACTION_DREAMING_STOPPED	scheduleSandmanLocked

在文件 PowerManagerService.java 中除了注册上述 5 个 Receiver 之外，还定义了一个 SettingsObserver 以监视系统中以下属性的变化。

☑　SCREENSAVER_ENABLE：屏保的功能开启。

☑　SCREENSAVER_ACTIVE_ON_SLEEP：在睡眠时屏保启动。

☑　SCREENSAVER_ACTIVE_ON_DOCK：连接底座并且屏保启动。

☑　SCREEN_OFF_TIMEOUT：休眠时间。

☑　STAY_ON_PLUGGED_IN：有插入并且屏幕开启。

☑　SCREEN_BRIGHTNESS：屏幕的亮度。

☑　SCREEN_BRIGHTNESS_MODE：屏幕亮度的模式。

SettingObserver 会监视到上述属性发生的变化，并且会调用 SettingObserver 中的 onChange()方法。

```
public void onChange(boolean selfChange, Uri uri) {
            synchronized (mLock) {
                handleSettingsChangedLocked();
            }
        }
```

由此可见，PowerManagerService 不但能够接收用户的请求，被动地去做一些操作，而且还要主动地监视

系统中一些重要的属性的变化和重要事件的发生。无论是处理主动还是被动的操作，在上面都一一列出了对应的处理函数。

4．分析核心函数

（1）进入休眠状态

函数 goToSleep()的功能是进入休眠状态，具体实现代码如下所示。

```
@Override // Binder call
    public void goToSleep(long eventTime, int reason) {
        if (eventTime > SystemClock.uptimeMillis()) {
            throw new IllegalArgumentException("event time must not be in the future");
        }
//权限检查
            mContext.enforceCallingOrSelfPermission(android.Manifest.permission.DEVICE_POWER, null);

        final long ident = Binder.clearCallingIdentity();
        try {
//这里会调用函数的实现，在 PowerManagerService 中有很多类似的使用方式，之后的代码中笔者会直接列出对应方法的实现
            goToSleepInternal(eventTime, reason);
        } finally {
            Binder.restoreCallingIdentity(ident);
        }
    }
    private void goToSleepInternal(long eventTime, int reason) {
        synchronized (mLock) {
            if (goToSleepNoUpdateLocked(eventTime, reason)) {
                updatePowerStateLocked();
            }
        }
    }
```

（2）确定是否需要休眠

函数 goToSleepNoUpdateLocked()的功能是确定是否要进入 goToSleep（休眠）状态，具体实现代码如下所示。

```
private boolean goToSleepNoUpdateLocked(long eventTime, int reason) {
    if (DEBUG_SPEW) {
        Slog.d(TAG, "goToSleepNoUpdateLocked: eventTime=" + eventTime + ", reason=" + reason);
    }

    if (eventTime < mLastWakeTime || mWakefulness == WAKEFULNESS_ASLEEP
            || !mBootCompleted || !mSystemReady) {
        return false;
    }

    switch (reason) {
        case PowerManager.GO_TO_SLEEP_REASON_DEVICE_ADMIN:
            Slog.i(TAG, "Going to sleep due to device administration policy...");
            break;
        case PowerManager.GO_TO_SLEEP_REASON_TIMEOUT:
            Slog.i(TAG, "Going to sleep due to screen timeout...");
            break;
```

```
        default:
            Slog.i(TAG, "Going to sleep by user request...");
            reason = PowerManager.GO_TO_SLEEP_REASON_USER;
            break;
    }

    sendPendingNotificationsLocked();
    mNotifier.onGoToSleepStarted(reason);
    mSendGoToSleepFinishedNotificationWhenReady = true;

    mLastSleepTime = eventTime;
    mDirty |= DIRTY_WAKEFULNESS;
    mWakefulness = WAKEFULNESS_ASLEEP;

    // Report the number of wake locks that will be cleared by going to sleep.
    int numWakeLocksCleared = 0;
    final int numWakeLocks = mWakeLocks.size();
    for (int i = 0; i < numWakeLocks; i++) {
        final WakeLock wakeLock = mWakeLocks.get(i);
        switch (wakeLock.mFlags & PowerManager.WAKE_LOCK_LEVEL_MASK) {
            case PowerManager.FULL_WAKE_LOCK:
            case PowerManager.SCREEN_BRIGHT_WAKE_LOCK:
            case PowerManager.SCREEN_DIM_WAKE_LOCK:
                numWakeLocksCleared += 1;
                break;
        }
    }
    EventLog.writeEvent(EventLogTags.POWER_SLEEP_REQUESTED, numWakeLocksCleared);
    return true;
}
```

通过上述代码可知，在此并没有真正地让设备进入 sleep 休眠状态，而只是把 PowerManagerService 中一些必要的属性进行了赋值处理。正因为如此，在 Android 系统中可以把多个 power state 属性的多个变化放在一起共同执行，而真正的功能执行者就是 updatePowerStateLocked。

注意： 在PowerManagerService的具体实现代码中，有很多含有xxxNoUpdateLocked格式后缀的函数名字，其实现原理类似于函数goToSleepNoUpdateLocked()。

（3）更新电源状态的锁定

函数 updatePowerStateLocked()的功能是更新电源状态的锁定，也就是把影响到 Power Management 发生变化的放在一起进行更新，让电源管理机制能够真正地起到作用。函数 updatePowerStateLocked()的具体实现代码如下所示。

```
private void updatePowerStateLocked() {
    if (!mSystemReady || mDirty == 0) {//如果系统没有准备好，或者 power state 没有发生任何变化，这个方法
可以不用执行
        return;
    }
    // Phase 0: Basic state updates.
    updateIsPoweredLocked(mDirty);
    updateStayOnLocked(mDirty);
    // Phase 1: Update wakefulness.
```

```
        // Loop because the wake lock and user activity computations are influenced
        // by changes in wakefulness.
        final long now = SystemClock.uptimeMillis();
        int dirtyPhase2 = 0;
        for (;;) {
            int dirtyPhase1 = mDirty;
            dirtyPhase2 |= dirtyPhase1;
            mDirty = 0;
//在前面解释几个变量时，就已经提到了 WakeLockSummary 和 UserActivitySummary
            updateWakeLockSummaryLocked(dirtyPhase1);
//在这里的两个方法中已经开始用到了。通过方法名，也可了解其功能
            updateUserActivitySummaryLocked(now, dirtyPhase1);
            if (!updateWakefulnessLocked(dirtyPhase1)) {
                break;
            }
        }
        // Phase 2: Update dreams and display power state.
        updateDreamLocked(dirtyPhase2);
        updateDisplayPowerStateLocked(dirtyPhase2);
        // Phase 3: Send notifications, if needed.
        if (mDisplayReady) {
            sendPendingNotificationsLocked();
        }
        // Phase 4: Update suspend blocker.
        // Because we might release the last suspend blocker here, we need to make sure
        // we finished everything else first!
        updateSuspendBlockerLocked();
}
```

通过上述实现代码可知，函数 updatePowerStateLocked()按照如下 4 个阶段对 Power State（电源状态）进行更新。

☑ 第 1 阶段：基本状态的更新。

首先执行函数 updateIsPoweredLocked()，功能是判断设备是否处于充电状态中，如果 DIRTY_BATTERY_STATE 发生了变化，说明设备的电池状态有过改变。函数 updateIsPoweredLocked()的具体实现代码如下所示。

```
/**
* Updates the value of mIsPowered.
* Sets DIRTY_IS_POWERED if a change occurred.
*/
private void updateIsPoweredLocked(int dirty) {
    if ((dirty & DIRTY_BATTERY_STATE) != 0) {
        final boolean wasPowered = mIsPowered;
        final int oldPlugType = mPlugType;
        mIsPowered = mBatteryService.isPowered(BatteryManager.BATTERY_PLUGGED_ANY);
        mPlugType = mBatteryService.getPlugType();
        mBatteryLevel = mBatteryService.getBatteryLevel();

        if (DEBUG) {
            Slog.d(TAG, "updateIsPoweredLocked: wasPowered=" + wasPowered
                    + ", mIsPowered=" + mIsPowered
                    + ", oldPlugType=" + oldPlugType
                    + ", mPlugType=" + mPlugType
```

```
                    + ", mBatteryLevel=" + mBatteryLevel);
        }

        if (wasPowered != mIsPowered || oldPlugType != mPlugType) {
            mDirty |= DIRTY_IS_POWERED;

            // Update wireless dock detection state.
            final boolean dockedOnWirelessCharger = mWirelessChargerDetector.update(
                    mIsPowered, mPlugType, mBatteryLevel);

            // Treat plugging and unplugging the devices as a user activity.
            // Users find it disconcerting when they plug or unplug the device
            // and it shuts off right away.
            // Some devices also wake the device when plugged or unplugged because
            // they don't have a charging LED.
            final long now = SystemClock.uptimeMillis();
            if (shouldWakeUpWhenPluggedOrUnpluggedLocked(wasPowered, oldPlugType,
                    dockedOnWirelessCharger)) {
                wakeUpNoUpdateLocked(now);
            }
            userActivityNoUpdateLocked(
                    now, PowerManager.USER_ACTIVITY_EVENT_OTHER, 0, Process.SYSTEM_UID);

            // Tell the notifier whether wireless charging has started so that
            // it can provide feedback to the user.
            if (dockedOnWirelessCharger) {
                mNotifier.onWirelessChargingStarted();
            }
        }
    }
}
```

然后通过对比判断（通过电池的状态前后的变化和充电状态的变化来判断）确定是否处于充电状态，会在 mDirty 中标记出充电方式的改变，并同时根据充电状态的变化进行一些相应的处理，同时在处理是否在充电或者充电方式的改变时，都会认为是发生了一次用户事件或者称为用户活动的发生。

接着执行函数 updateStayOnLocked()，功能是更新 device 是否处于开启状态。此函数也是通过 mStayOn 发生的前后变化作为判断依据，如果 device 的属性 Settings.Global.STAY_ON_WHILE_ PLUGGED_IN 为置位，并且没有达到电池充电时持续开屏时间的最大值（也就是说，在插入电源后的一段时间内保持开屏状态），那么 mStayOn 为真。函数 updateStayOnLocked()的具体实现代码如下所示。

```
/**
 * Updates the value of mStayOn.
 * Sets DIRTY_STAY_ON if a change occurred.
 */
private void updateStayOnLocked(int dirty) {
    if ((dirty & (DIRTY_BATTERY_STATE | DIRTY_SETTINGS)) != 0) {
        final boolean wasStayOn = mStayOn;
        if (mStayOnWhilePluggedInSetting != 0
                && !isMaximumScreenOffTimeoutFromDeviceAdminEnforcedLocked()) {
            mStayOn = mBatteryService.isPowered(mStayOnWhilePluggedInSetting);
        } else {
            mStayOn = false;
```

```
            }

            if (mStayOn != wasStayOn) {
                mDirty |= DIRTY_STAY_ON;
            }
        }
    }
```

由此可见，在第 1 阶段的更新过程中主要是进行了充电状态的判断工作，然后根据充电的状态更新了一些必要的属性变化，同时更新了 mDirty。

☑ 第 2 阶段：显示内容的更新。

mWakefulness 表示 device 处于醒着或睡眠或两者之间的一种状态，这种状态会影响到 wake lock 和 user activity 的计算，所以要进行更新。在第 2 阶段先通过一个死循环进行处理，只有当 updateWakefulnessLocked 返回为 false 时才能跳出这个循环。在刚刚进入这个循环时，对 mDirty 进行了重置，说明在这次 updatePowerState 后会执行前面所有发生的 power state，而不会让其影响到下一次的变化。同时也在为下一次的 power state 从头开始更新做好准备。其中函数 updateWakeLockSummaryLocked()的实现代码如下所示。

```
private void updateWakeLockSummaryLocked(int dirty) {
    if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_WAKEFULNESS)) != 0) {
        mWakeLockSummary = 0;

        final int numWakeLocks = mWakeLocks.size();
        for (int i = 0; i < numWakeLocks; i++) {
            final WakeLock wakeLock = mWakeLocks.get(i);
            switch (wakeLock.mFlags & PowerManager.WAKE_LOCK_LEVEL_MASK) {
                case PowerManager.PARTIAL_WAKE_LOCK:
                    mWakeLockSummary |= WAKE_LOCK_CPU;
                    break;
                case PowerManager.FULL_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |= WAKE_LOCK_CPU
                                | WAKE_LOCK_SCREEN_BRIGHT | WAKE_LOCK_BUTTON_BRIGHT;
                        if (mWakefulness == WAKEFULNESS_AWAKE) {
                            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
                        }
                    }
                    break;
                case PowerManager.SCREEN_BRIGHT_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |= WAKE_LOCK_CPU | WAKE_LOCK_SCREEN_BRIGHT;
                        if (mWakefulness == WAKEFULNESS_AWAKE) {
                            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
                        }
                    }
                    break;
                case PowerManager.SCREEN_DIM_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |= WAKE_LOCK_CPU | WAKE_LOCK_SCREEN_DIM;
                        if (mWakefulness == WAKEFULNESS_AWAKE) {
                            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
                        }
                    }
```

```
                    break;
                case PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary  |=  WAKE_LOCK_CPU  |  WAKE_LOCK_PROXIMITY_
SCREEN_OFF;
                    }
                    break;
            }
        }

        if (DEBUG_SPEW) {
            Slog.d(TAG, "updateWakeLockSummaryLocked: mWakefulness="
                    + wakefulnessToString(mWakefulness)
                    + ", mWakeLockSummary=0x" + Integer.toHexString(mWakeLockSummary));
        }
    }
}
```

函数 updateUserActivitySummaryLocked()对关屏时间和变暗时间进行了比较。假设在系统中设置的睡眠时间是 30 秒，而在 PowerManagerService 中默认的 SCREEN_DIM_DURATION 是 7 秒，则说明如果没有用户活动的话，则设备的屏幕在第 23 秒开始变换，持续 7 秒时间，然后才开始关闭屏幕。函数 updateUserActivitySummaryLocked 的具体实现代码如下所示。

```
private void updateUserActivitySummaryLocked(long now, int dirty) {
    // Update the status of the user activity timeout timer.
    if ((dirty & (DIRTY_USER_ACTIVITY | DIRTY_WAKEFULNESS | DIRTY_SETTINGS)) != 0) {
        mHandler.removeMessages(MSG_USER_ACTIVITY_TIMEOUT);
        long nextTimeout = 0;
        if (mWakefulness != WAKEFULNESS_ASLEEP) {
            final int screenOffTimeout = getScreenOffTimeoutLocked();
            final int screenDimDuration = getScreenDimDurationLocked(screenOffTimeout);
            mUserActivitySummary = 0;
            if (mLastUserActivityTime >= mLastWakeTime) {
                nextTimeout = mLastUserActivityTime
                        + screenOffTimeout - screenDimDuration;
                if (now < nextTimeout) {
                    mUserActivitySummary |= USER_ACTIVITY_SCREEN_BRIGHT;
                } else {
                    nextTimeout = mLastUserActivityTime + screenOffTimeout;
                    if (now < nextTimeout) {
                        mUserActivitySummary |= USER_ACTIVITY_SCREEN_DIM;
                    }
                }
            }
            if (mUserActivitySummary == 0
                    && mLastUserActivityTimeNoChangeLights >= mLastWakeTime) {
                nextTimeout = mLastUserActivityTimeNoChangeLights + screenOffTimeout;
                if (now < nextTimeout
                        && mDisplayPowerRequest.screenState
                            != DisplayPowerRequest.SCREEN_STATE_OFF) {
                    mUserActivitySummary = mDisplayPowerRequest.screenState
                            == DisplayPowerRequest.SCREEN_STATE_BRIGHT ?
```

```
                              USER_ACTIVITY_SCREEN_BRIGHT : USER_ACTIVITY_SCREEN_DIM;
                    }
                }
                if (mUserActivitySummary != 0) {
                    Message msg = mHandler.obtainMessage(MSG_USER_ACTIVITY_TIMEOUT);
                    msg.setAsynchronous(true);
                    mHandler.sendMessageAtTime(msg, nextTimeout);
                }
            } else {
                mUserActivitySummary = 0;
            }
            if (DEBUG_SPEW) {
                Slog.d(TAG, "updateUserActivitySummaryLocked: mWakefulness="
                        + wakefulnessToString(mWakefulness)
                        + ", mUserActivitySummary=0x" + Integer.toHexString(mUserActivitySummary)
                        + ", nextTimeout=" + TimeUtils.formatUptime(nextTimeout));
            }
        }
    }
```

具体在什么时候才可以跳出这个循环，需要视函数 updateWakefulnessLocked()的返回值而定。函数updateWakefulnessLocked()的具体实现代码如下所示。

```
/**
     * 这个方法的功能是：根据当前的 wakeLocks 和用户的活动情况，来决定设备是否需要休眠
     *
        *///当 wakefulness 发生变化时，返回 true，同时也需要重新计算 power state
    private boolean updateWakefulnessLocked(int dirty) {
        boolean changed = false;
        if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_BOOT_COMPLETED
                | DIRTY_WAKEFULNESS | DIRTY_STAY_ON | DIRTY_PROXIMITY_POSITIVE
                | DIRTY_DOCK_STATE)) != 0) {
            if (mWakefulness == WAKEFULNESS_AWAKE && isItBedTimeYetLocked()) {
                if (DEBUG_SPEW) {
                    Slog.d(TAG, "updateWakefulnessLocked: Bed time...");
                }
                final long time = SystemClock.uptimeMillis();
                if (shouldNapAtBedTimeLocked()) {
                    changed = napNoUpdateLocked(time);
                } else {
                    changed = goToSleepNoUpdateLocked(time,
                            PowerManager.GO_TO_SLEEP_REASON_TIMEOUT);
                }
            }
        }
        return changed;
    }
```

在上述代码中用到了函数 isItBedTimeYetLocked()，功能是询问是否到了应该休眠的时间，如果现在设备处于醒着的状态，则马上改变为睡眠时间。函数 isItBedTimeYetLocked()的具体实现代码如下所示。

```
private boolean isItBedTimeYetLocked() {
    return mBootCompleted && !isBeingKeptAwakeLocked();
}
```

在上述代码中，如果有应用程序持有 wakelock 或产生了用户活动，或者处于充电状态，那么 isBeingKeptAwakeLocked()的返回值就是 true，相应地，isItBedTimeYetLocked 返回值为 false，因为还有没释放的 wakelock，或者有用户活动或者是在充电等，则说明还没有到睡眠的时间。但是，如果 wakelock 都释放了，并且也没有用户活动了，那么就可以进入睡眠状态。这时的设备由醒着状态转换为睡眠的处理过程，此过程需要调用函数 updateWakefulnessLocked()实现，此函数的具体实现代码如下所示。

```
private boolean shouldNapAtBedTimeLocked() {
    return mDreamsActivateOnSleepSetting
            || (mDreamsActivateOnDockSetting
                    && mDockState != Intent.EXTRA_DOCK_STATE_UNDOCKED);
}
```

在上述代码中，mDreamsActivateOnSleepSetting 的默认值为 false，mDreamsActivateOnDockSetting 的默认值为 true。因为一般的用户没有接入 Dock，mDockState != Intent.EXTRA_DOCK_STATE_UNDOCKED 的值为 false，所以上述函数的返回值是 false。这样接下来需要执行函数 goToSleepNoUpdateLocked()，此函数已经在前面进行了讲解，此函数只是更新了 power state 中一些必要的属性，并没有进行真正的执行能够让 Device 进入 sleep 的代码，真正的执行代码在函数 updatePowerStateLocked()中。

☑　第 3 阶段：dream 和 display 状态的更新。

在这一阶段，函数 updateDreamLocked()会根据 mDirty 的变化，并结合其他的属性共同判断是否要开始屏保（Dreaming）处理。如果需要开始进行屏保处理，需要通过 DreamManagerService 开启 Dreaming。函数 updateDreamLocked()的具体实现代码如下所示。

```
private void updateDreamLocked(int dirty) {
    if ((dirty & (DIRTY_WAKEFULNESS
            | DIRTY_USER_ACTIVITY
            | DIRTY_WAKE_LOCKS
            | DIRTY_BOOT_COMPLETED
            | DIRTY_SETTINGS
            | DIRTY_IS_POWERED
            | DIRTY_STAY_ON
            | DIRTY_PROXIMITY_POSITIVE
            | DIRTY_BATTERY_STATE)) != 0) {
        scheduleSandmanLocked();
    }
}
```

函数 updateDisplayPowerStateLocked()的主要功能是每次重新计算 Display Power State 的值，即 SCREEN_STATE_OFF、SCREEN_STATE_DIM 和 SCREEN_STATE_BRIGHT 这 3 个值之一。如果在 DisplayController 中更新了 Display Power State 的值，那么 DisplayController 会发送通知消息，所以还需要回来重新检查一次。函数 updateDisplayPowerStateLocked()的具体实现代码如下所示。

```
private void updateDisplayPowerStateLocked(int dirty) {
        if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_WAKEFULNESS
                | DIRTY_ACTUAL_DISPLAY_POWER_STATE_UPDATED | DIRTY_BOOT_COMPLETED
                | DIRTY_SETTINGS | DIRTY_SCREEN_ON_BLOCKER_RELEASED)) != 0) {
            //获取 display 的 power state 将要变成的状态
            int newScreenState = getDesiredScreenPowerStateLocked();
            if (newScreenState != mDisplayPowerRequest.screenState) {//mDisplayPowerRequest.screenState
是目前 display 所处的 power state
                if (newScreenState == DisplayPowerRequest.SCREEN_STATE_OFF
                        && mDisplayPowerRequest.screenState
```

```
                                //这个判断意味着：目前 display 的电源状态不是 OFF，但是想要变为 OFF
                                != DisplayPowerRequest.SCREEN_STATE_OFF) {
                        mLastScreenOffEventElapsedRealTime = SystemClock.elapsedRealtime();
                    }
                    mDisplayPowerRequest.screenState = newScreenState;
                    nativeSetPowerState(
                            newScreenState != DisplayPowerRequest.SCREEN_STATE_OFF,
                            newScreenState == DisplayPowerRequest.SCREEN_STATE_BRIGHT);
                }
                int screenBrightness = mScreenBrightnessSettingDefault;
                float screenAutoBrightnessAdjustment = 0.0f;
                boolean autoBrightness = (mScreenBrightnessModeSetting ==
                        //获取屏幕亮度模式是否为自动变化
                        Settings.System.SCREEN_BRIGHTNESS_MODE_AUTOMATIC);
                if (isValidBrightness(mScreenBrightnessOverrideFromWindowManager))
{//mScreenBrightnessOverideFromeWindowManager 是 WindowManager 设置的亮度大小，默认值为-1
                    screenBrightness = mScreenBrightnessOverrideFromWindowManager;
                    autoBrightness = false;
                } else if (isValidBrightness(mTemporaryScreenBrightnessSettingOverride))
{//mTemporaryScreenBrightnessSettingOverride 在 widget 中设置的临时亮度大小，默认为-1
                    screenBrightness = mTemporaryScreenBrightnessSettingOverride;
                } else if (isValidBrightness(mScreenBrightnessSetting)) {//在 Settings 中设置的默认亮度，在 Android 4.2
中其值为 102
                    screenBrightness = mScreenBrightnessSetting;
                }
                if (autoBrightness) {//如果亮度是自动调节的话
                    screenBrightness = mScreenBrightnessSettingDefault;
                    if (isValidAutoBrightnessAdjustment(
                            mTemporaryScreenAutoBrightnessAdjustmentSettingOverride)) {
                        screenAutoBrightnessAdjustment =
                                mTemporaryScreenAutoBrightnessAdjustmentSettingOverride;
                    } else if (isValidAutoBrightnessAdjustment(
                            mScreenAutoBrightnessAdjustmentSetting)) {
                        screenAutoBrightnessAdjustment = mScreenAutoBrightnessAdjustmentSetting;
                    }
                }
                screenBrightness = Math.max(Math.min(screenBrightness,
                        mScreenBrightnessSettingMaximum), mScreenBrightnessSettingMinimum);
                screenAutoBrightnessAdjustment = Math.max(Math.min(
                        screenAutoBrightnessAdjustment, 1.0f), -1.0f);
                //从这行向下开始就是配置完成 DisplayPowerRequest，然后以此为参数通过 requestPowerSTate
方法进行设置
                mDisplayPowerRequest.screenBrightness = screenBrightness;
                mDisplayPowerRequest.screenAutoBrightnessAdjustment =
                        screenAutoBrightnessAdjustment;
                mDisplayPowerRequest.useAutoBrightness = autoBrightness;
                mDisplayPowerRequest.useProximitySensor = shouldUseProximitySensorLocked();
                mDisplayPowerRequest.blockScreenOn = mScreenOnBlocker.isHeld();
                mDisplayReady = mDisplayPowerController.requestPowerState(mDisplayPowerRequest,
                        mRequestWaitForNegativeProximity);
                mRequestWaitForNegativeProximity = false;
```

```
                if (DEBUG_SPEW) {
                    Slog.d(TAG, "updateScreenStateLocked: mDisplayReady=" + mDisplayReady
                            + ", newScreenState=" + newScreenState
                            + ", mWakefulness=" + mWakefulness
                            + ", mWakeLockSummary=0x" + Integer.toHexString(mWakeLockSummary)
                            + ", mUserActivitySummary=0x" + Integer.toHexString(mUserActivitySummary)
                            + ", mBootCompleted=" + mBootCompleted);
                }
            }
        }
```

在上述代码中实现了对屏幕亮度的请求，并改变了屏幕的亮度。在调用的函数 goToSleepNoUpdateLocked()中，Display State 会因为 requestPowerState()函数的调用而起作用。

☑　第 4 阶段：suspend blocker 的更新。

之所以放在最后一步才进行 Suspend Blocker（暂停阻滞）的更新，是因为在这里可能会释放 Suspend Blocker。本阶段 Suspend Blocker 的更新很简单，仅需要判断 Device 是否需要持有 CPU 或者是否需要 CPU 继续运行，如果有没有释放的 WakeLock，或者还有用户活动，或者屏幕没有关闭等，则肯定是需要持有 CPU。所以这里就是根据需求去申请或者释放 SuspendBlocker。

到此为止，整个 PowerManagerService 的工作过程介绍完毕，已经基本上讲解了 PowerManagerService 在 Framework 中的实现过程。

（4）更新状态

如果在屏幕中点击或者滑动一次就会调用一次 userActivity()函数，该函数会更新一些状态，其中比较重要的是 mUserState 值的状态，此值决定了系统对 LCD、按键以及键盘的亮和灭，例如 mUserState = 0X7 表示 LCD 和按键背光亮。函数 userActivity()的具体实现代码如下所示。

```
public void userActivity(long eventTime, int event, int flags) {
    final long now = SystemClock.uptimeMillis();
    if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.DEVICE_POWER)
            != PackageManager.PERMISSION_GRANTED) {
        // Once upon a time applications could call userActivity().
        // Now we require the DEVICE_POWER permission. Log a warning and ignore the
        // request instead of throwing a SecurityException so we don't break old apps.
        synchronized (mLock) {
            if (now >= mLastWarningAboutUserActivityPermission + (5 * 60 * 1000)) {
                mLastWarningAboutUserActivityPermission = now;
                Slog.w(TAG, "Ignoring call to PowerManager.userActivity() because the "
                        + "caller does not have DEVICE_POWER permission.   "
                        + "Please fix your app!   "
                        + " pid=" + Binder.getCallingPid()
                        + " uid=" + Binder.getCallingUid());
            }
        }
        return;
    }
     if (eventTime > SystemClock.uptimeMillis()) {
        throw new IllegalArgumentException("event time must not be in the future");
    }
    final int uid = Binder.getCallingUid();
    final long ident = Binder.clearCallingIdentity();
    try {
```

```
            userActivityInternal(eventTime, event, flags, uid);
        } finally {
            Binder.restoreCallingIdentity(ident);
        }
}
```

（5）获得唤醒锁

函数 acquireWakeLock()的功能是获得唤醒锁，文件 PowerManagerService.java 的最基本功能是管理所有的应用程序申请的 wakelock。函数 acquireWakeLock()的具体实现代码如下所示。

```
public void acquireWakeLock(IBinder lock, int flags, String tag, WorkSource ws) {
    if (lock == null) {
        throw new IllegalArgumentException("lock must not be null");
    }
    PowerManager.validateWakeLockParameters(flags, tag);

    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.WAKE_LOCK, null);
    if (ws != null && ws.size() != 0) {
        mContext.enforceCallingOrSelfPermission(
                android.Manifest.permission.UPDATE_DEVICE_STATS, null);
    } else {
        ws = null;
    }

    final int uid = Binder.getCallingUid();
    final int pid = Binder.getCallingPid();
    final long ident = Binder.clearCallingIdentity();
    try {
        acquireWakeLockInternal(lock, flags, tag, ws, uid, pid);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

在 Android 系统中，音频播放器、视频播放器、Camera 等申请的 wakelock 都是通过这个类来管理的。下面的代码调用了类 Power 中的函数 acquireWakeLock()，此时的 PARTIAL_NAME 作为参数传递到底层上。

```
static final String PARTIAL_NAME = "PowerManagerService"
Power.acquireWakeLock(Power.PARTIAL_WAKE_LOCK,
        PARTIAL_NAME);
```

（6）获得唤醒的内部锁

函数 acquireWakeLockInternal()的功能是获得唤醒的内部锁，具体实现代码如下所示。

```
private void acquireWakeLockInternal(IBinder lock, int flags, String tag, WorkSource ws,
        int uid, int pid) {
    synchronized (mLock) {
        if (DEBUG_SPEW) {
            Slog.d(TAG, "acquireWakeLockInternal: lock=" + Objects.hashCode(lock)
                    + ", flags=0x" + Integer.toHexString(flags)
                    + ", tag=\"" + tag + "\", ws=" + ws + ", uid=" + uid + ", pid=" + pid);
        }

        WakeLock wakeLock;
        int index = findWakeLockIndexLocked(lock);
```

```
            if (index >= 0) {
                wakeLock = mWakeLocks.get(index);
                if (!wakeLock.hasSameProperties(flags, tag, ws, uid, pid)) {
                    // Update existing wake lock.   This shouldn't happen but is harmless.
                    notifyWakeLockReleasedLocked(wakeLock);
                    wakeLock.updateProperties(flags, tag, ws, uid, pid);
                    notifyWakeLockAcquiredLocked(wakeLock);
                }
            } else {
                wakeLock = new WakeLock(lock, flags, tag, ws, uid, pid);
                try {
                    lock.linkToDeath(wakeLock, 0);
                } catch (RemoteException ex) {
                    throw new IllegalArgumentException("Wake lock is already dead.");
                }
                notifyWakeLockAcquiredLocked(wakeLock);
                mWakeLocks.add(wakeLock);
           }

            applyWakeLockFlagsOnAcquireLocked(wakeLock);
            mDirty |= DIRTY_WAKE_LOCKS;
            updatePowerStateLocked();
        }
}
```

（7）获取锁中标志的唤醒锁

函数 applyWakeLockFlagsOnAcquireLocked()的功能是在获取的锁中标志某个唤醒锁，具体实现代码如下所示。

```
private void applyWakeLockFlagsOnAcquireLocked(WakeLock wakeLock) {
    if ((wakeLock.mFlags & PowerManager.ACQUIRE_CAUSES_WAKEUP) != 0 &&
                isScreenLock(wakeLock)) {
            wakeUpNoUpdateLocked(SystemClock.uptimeMillis());
    }
}
```

（8）释放唤醒锁

函数 releaseWakeLock()的功能是释放唤醒锁，具体实现代码如下所示。

```
public void releaseWakeLock(IBinder lock, int flags) {
    if (lock == null) {
        throw new IllegalArgumentException("lock must not be null");
    }
    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.WAKE_LOCK, null);
    final long ident = Binder.clearCallingIdentity();
    try {
        releaseWakeLockInternal(lock, flags);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

（9）释放唤醒内部锁

函数 releaseWakeLockInternal()的功能是释放唤醒内部锁，具体实现代码如下所示。

```
private void releaseWakeLockInternal(IBinder lock, int flags) {
    synchronized (mLock) {
        int index = findWakeLockIndexLocked(lock);
        if (index < 0) {
            if (DEBUG_SPEW) {
                Slog.d(TAG, "releaseWakeLockInternal: lock=" + Objects.hashCode(lock)
                        + " [not found], flags=0x" + Integer.toHexString(flags));
            }
            return;
        }

        WakeLock wakeLock = mWakeLocks.get(index);
        if (DEBUG_SPEW) {
            Slog.d(TAG, "releaseWakeLockInternal: lock=" + Objects.hashCode(lock)
                    + " [" + wakeLock.mTag + "], flags=0x" + Integer.toHexString(flags));
        }

        mWakeLocks.remove(index);
        notifyWakeLockReleasedLocked(wakeLock);
        wakeLock.mLock.unlinkToDeath(wakeLock, 0);

        if ((flags & PowerManager.WAIT_FOR_PROXIMITY_NEGATIVE) != 0) {
            mRequestWaitForNegativeProximity = true;
        }

        applyWakeLockFlagsOnReleaseLocked(wakeLock);
        mDirty |= DIRTY_WAKE_LOCKS;
        updatePowerStateLocked();
    }
}
```

（10）更新唤醒锁资源和内部唤醒锁的资源

函数 updateWakeLockWorkSource()和 updateWakeLockWorkSourceInternal()被绑定机制 Binder 所调用，功能是分别更新唤醒锁资源和内部唤醒锁的资源，这两个函数的具体实现代码如下所示。

```
public void updateWakeLockWorkSource(IBinder lock, WorkSource ws) {
    if (lock == null) {
        throw new IllegalArgumentException("lock must not be null");
    }

    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.WAKE_LOCK, null);
    if (ws != null && ws.size() != 0) {
        mContext.enforceCallingOrSelfPermission(
                android.Manifest.permission.UPDATE_DEVICE_STATS, null);
    } else {
        ws = null;
    }

    final long ident = Binder.clearCallingIdentity();
    try {
        updateWakeLockWorkSourceInternal(lock, ws);
    } finally {
```

```
            Binder.restoreCallingIdentity(ident);
        }
    }

private void updateWakeLockWorkSourceInternal(IBinder lock, WorkSource ws) {
    synchronized (mLock) {
        int index = findWakeLockIndexLocked(lock);
        if (index < 0) {
            if (DEBUG_SPEW) {
                Slog.d(TAG, "updateWakeLockWorkSourceInternal: lock=" + Objects.hashCode(lock)
                        + " [not found], ws=" + ws);
            }
            throw new IllegalArgumentException("Wake lock not active");
        }

        WakeLock wakeLock = mWakeLocks.get(index);
        if (DEBUG_SPEW) {
            Slog.d(TAG, "updateWakeLockWorkSourceInternal: lock=" + Objects.hashCode(lock)
                    + " [" + wakeLock.mTag + "], ws=" + ws);
        }

        if (!wakeLock.hasSameWorkSource(ws)) {
            notifyWakeLockReleasedLocked(wakeLock);
            wakeLock.updateWorkSource(ws);
            notifyWakeLockAcquiredLocked(wakeLock);
        }
    }
}
```

10.3　JNI 层架构分析

在 Android 的电源管理系统中，Native 申明在 Power 类中并没有实现的方法，而是在文件 frameworks/base/core/jni/android_os_Power.cpp 中实现的。也就是说，文件 android_os_Power.cpp 就是 Power Management 系统的 JNI 层。本节将详细分析 Android 电源管理系统中的 JNI 层的基本架构知识。

10.3.1　定义两层之间的接口函数

文件 android_os_Power.cpp 比较简单，定义了连接应用层和内核层之间的桥梁作用的接口函数，这些函数已经在本章前面的 Framework 层部分的内容中调用过。文件 android_os_Power.cpp 的主要实现代码如下所示。

```
static void
acquireWakeLock(JNIEnv *env, jobject clazz, jint lock, jstring idObj)
{
    if (idObj == NULL) {
        jniThrowNullPointerException(env, "id is null");
        return ;
    }
    const char *id = env->GetStringUTFChars(idObj, NULL);
```

```
    acquire_wake_lock(lock, id);
    env->ReleaseStringUTFChars(idObj, id);
}
static void
releaseWakeLock(JNIEnv *env, jobject clazz, jstring idObj)
{
    if (idObj == NULL) {
        jniThrowNullPointerException(env, "id is null");
        return ;
    }
    const char *id = env->GetStringUTFChars(idObj, NULL);
    release_wake_lock(id);
    env->ReleaseStringUTFChars(idObj, id);
}
static int
setLastUserActivityTimeout(JNIEnv *env, jobject clazz, jlong timeMS)
{
    return set_last_user_activity_timeout(timeMS/1000);
}
static int
setScreenState(JNIEnv *env, jobject clazz, jboolean on)
{
    return set_screen_state(on);
}
static void android_os_Power_shutdown(JNIEnv *env, jobject clazz)
{
    android_reboot(ANDROID_RB_POWEROFF, 0, 0);
}
static void android_os_Power_reboot(JNIEnv *env, jobject clazz, jstring reason)
{
    if (reason == NULL) {
        android_reboot(ANDROID_RB_RESTART, 0, 0);
    } else {
        const char *chars = env->GetStringUTFChars(reason, NULL);
        android_reboot(ANDROID_RB_RESTART2, 0, (char *) chars);
        env->ReleaseStringUTFChars(reason, chars);    // In case it fails.
    }
    jniThrowIOException(env, errno);
}
static JNINativeMethod method_table[] = {
    { "acquireWakeLock", "(ILjava/lang/String;)V", (void*)acquireWakeLock },
    { "releaseWakeLock", "(Ljava/lang/String;)V", (void*)releaseWakeLock },
    { "setLastUserActivityTimeout", "(J)I", (void*)setLastUserActivityTimeout },
    { "setScreenState", "(Z)I", (void*)setScreenState },
    { "shutdown", "()V", (void*)android_os_Power_shutdown },
    { "rebootNative", "(Ljava/lang/String;)V", (void*)android_os_Power_reboot },
};
int register_android_os_Power(JNIEnv *env)
{
    return AndroidRuntime::registerNativeMethods(
        env, "android/os/Power",
```

```
        method_table, NELEM(method_table));
}
};
```

10.3.2　与 Linux Kernel 层进行交互

与 Linux Kernel 的交互工作是通过文件 hardware/libhardware/power/power.c 实现的，Android 和 Kernel 的交互主要是通过 sys 文件的方式来实现的。文件 power.c 的具体实现代码如下所示。

```
static void power_init(struct power_module *module)
{
}
static void power_set_interactive(struct power_module *module, int on)
{
}
static void power_hint(struct power_module *module, power_hint_t hint,
                        void *data) {
    switch (hint) {
    default:
        break;
    }
}
static struct hw_module_methods_t power_module_methods = {
    .open = NULL,
};
struct power_module HAL_MODULE_INFO_SYM = {
    .common = {
        .tag = HARDWARE_MODULE_TAG,
        .module_api_version = POWER_MODULE_API_VERSION_0_2,
        .hal_api_version = HARDWARE_HAL_API_VERSION,
        .id = POWER_HARDWARE_MODULE_ID,
        .name = "Default Power HAL",
        .author = "The Android Open Source Project",
        .methods = &power_module_methods,
    },
    .init = power_init,
    .setInteractive = power_set_interactive,
    .powerHint = power_hint,
};
```

10.4　Kernel（内核）层架构分析

在 Android 系统中，Power Management 系统内核层的实现文件如下。

☑　drivers/android/power.c

☑　drivers/base/main.c

☑　drivers/base/power/suspend.c

☑　drivers/base/power/resume.c

- ☑ kernel/power/main.c
- ☑ kernel/power/wakelock.c
- ☑ kernel/power/unwakelock.c
- ☑ kernel/power/earlysuspend.c

本节将对上述内核层文件的具体实现进行详细讲解，为了节省本书的篇幅，只讲解主要的实现文件的核心代码。

10.4.1 文件 power.c

在 Power Management 系统的内核层中，实现文件 drivers/android/power.c 对 Kernel 提供了如下接口函数。

（1）初始化 Suspend lock

接口函数 EXPORT_SYMBOL(android_init_suspend_lock)的功能是初始化 Suspend lock，在使用 Power Management 内核之前必须做初始化工作，函数 android_init_suspend_lock()的具体实现代码如下所示。

```
int android_init_suspend_lock(android_suspend_lock_t *lock)
{
	return android_init_suspend_lock_internal(lock, 0);
}
```

（2）释放 Suspend lock 相关的资源

接口函数 EXPORT_SYMBOL(android_uninit_suspend_lock)的功能是释放 suspend lock 相关的资源，具体实现代码如下所示。

```
void android_uninit_suspend_lock(android_suspend_lock_t *lock)
{
	unsigned long irqflags;
	if (android_power_debug_mask & ANDROID_POWER_DEBUG_WAKE_LOCK)
		printk(KERN_INFO "android_uninit_suspend_lock name=%s\n",
			lock->name);
	spin_lock_irqsave(&g_list_lock, irqflags);
#ifdef CONFIG_ANDROID_POWER_STAT
	if(lock->stat.count) {
		if(g_deleted_wake_locks.stat.count == 0) {
			g_deleted_wake_locks.name = "deleted_wake_locks";
			android_init_suspend_lock_internal(
				&g_deleted_wake_locks, 1);
		}
		g_deleted_wake_locks.stat.count += lock->stat.count;
		g_deleted_wake_locks.stat.expire_count += lock->stat.expire_count;
		g_deleted_wake_locks.stat.total_time = ktime_add(g_deleted_wake_locks.stat.total_time, lock->stat.
total_time);
		g_deleted_wake_locks.stat.max_time = ktime_add(g_deleted_wake_locks.stat.max_time, lock->stat.
max_time);
	}
#endif
	list_del(&lock->link);
	spin_unlock_irqrestore(&g_list_lock, irqflags);
}
```

（3）申请 lock 并调用相应的 unlock

接口函数 EXPORT_SYMBOL(android_lock_suspend)的功能是申请 lock，并调用相应的 unlock 来释放



lock。函数 android_lock_suspend()的具体实现代码如下所示。

```
void android_lock_suspend(android_suspend_lock_t *lock)
{
	unsigned long irqflags;
	spin_lock_irqsave(&g_list_lock, irqflags);
#ifdef CONFIG_ANDROID_POWER_STAT
	if(!(lock->flags & ANDROID_SUSPEND_LOCK_ACTIVE)) {
		lock->flags |= ANDROID_SUSPEND_LOCK_ACTIVE;
		lock->stat.last_time = ktime_get();
	}
#endif
	if (android_power_debug_mask & ANDROID_POWER_DEBUG_WAKE_LOCK)
		printk(KERN_INFO "android_power: acquire wake lock: %s\n",
			lock->name);
	lock->expires = INT_MAX;
	lock->flags &= ~ANDROID_SUSPEND_LOCK_AUTO_EXPIRE;
	list_del(&lock->link);
	list_add(&lock->link, &g_active_partial_wake_locks);
	g_current_event_num++;
	spin_unlock_irqrestore(&g_list_lock, irqflags);
}
```

（4）申请 Partial Wakelock

接口函数 EXPORT_SYMBOL(android_lock_suspend_auto_expire)的功能是申请 Partial Wakelock，当定时时间到后会自动释放。函数 android_lock_suspend_auto_expire()的具体实现代码如下所示。

```
void android_lock_suspend_auto_expire(android_suspend_lock_t *lock, int timeout)
{
	unsigned long irqflags;
	spin_lock_irqsave(&g_list_lock, irqflags);
#ifdef CONFIG_ANDROID_POWER_STAT
	if(!(lock->flags & ANDROID_SUSPEND_LOCK_ACTIVE)) {
		lock->flags |= ANDROID_SUSPEND_LOCK_ACTIVE;
		lock->stat.last_time = ktime_get();
	}
#endif
	if (android_power_debug_mask & ANDROID_POWER_DEBUG_WAKE_LOCK)
		printk(KERN_INFO "android_power: acquire wake lock: %s, "
			"timeout %d.%03lu\n", lock->name, timeout / HZ,
			(timeout % HZ) * MSEC_PER_SEC / HZ);
	lock->expires = jiffies + timeout;
	lock->flags |= ANDROID_SUSPEND_LOCK_AUTO_EXPIRE;
	list_del(&lock->link);
	list_add(&lock->link, &g_active_partial_wake_locks);
	g_current_event_num++;
	wake_up(&g_wait_queue);
	spin_unlock_irqrestore(&g_list_lock, irqflags);
}
```

（5）释放 lock

接口函数 EXPORT_SYMBOL(android_unlock_suspend)的功能是释放 lock，具体实现代码如下所示。

```
static void android_unlock_suspend_stat_locked(android_suspend_lock_t *lock)
{
```

```
	if(lock->flags & ANDROID_SUSPEND_LOCK_ACTIVE) {
		ktime_t duration;
		lock->flags &= ~ANDROID_SUSPEND_LOCK_ACTIVE;
		lock->stat.count++;
		duration = ktime_sub(ktime_get(), lock->stat.last_time);
		lock->stat.total_time = ktime_add(lock->stat.total_time, duration);
		if(ktime_to_ns(duration) > ktime_to_ns(lock->stat.max_time))
			lock->stat.max_time = duration;
		lock->stat.last_time = ktime_get();
	}
}
#endif
```

（6）注册 Early Suspend 驱动

接口函数 EXPORT_SYMBOL(android_register_early_suspend)的功能是注册 Early Suspend 的驱动，具体实现代码如下所示。

```
void android_register_early_suspend(android_early_suspend_t *handler)
{
	struct list_head *pos;

	mutex_lock(&g_early_suspend_lock);
	list_for_each(pos, &g_early_suspend_handlers) {
		android_early_suspend_t *e = list_entry(pos, android_early_suspend_t, link);
		if(e->level > handler->level)
			break;
	}
	list_add_tail(&handler->link, pos);
	mutex_unlock(&g_early_suspend_lock);
}
```

（7）取消已经注册的 Early Suspend 驱动

接口函数 EXPORT_SYMBOL(android_unregister_early_suspend)的功能是取消已经注册的 Early Suspend 的驱动，具体实现代码如下所示。

```
void android_unregister_early_suspend(android_early_suspend_t *handler)
{
	mutex_lock(&g_early_suspend_lock);
	list_del(&handler->link);
	mutex_unlock(&g_early_suspend_lock);
}
```

10.4.2 文件 earlysuspend.c

在 Power Management 系统的内核层中，实现文件 kernel/power/earlysuspend.c 对 Kernel 提供了如下所示的接口函数。

（1）注册 Early Suspend 驱动

接口函数 EXPORT_SYMBOL(register_early_suspend)的功能是注册 Early Suspend 的驱动，具体实现代码如下所示。

```
void register_early_suspend(struct early_suspend *handler)
{
		struct list_head *pos;
```

```
        mutex_lock(&early_suspend_lock);
        list_for_each(pos, &early_suspend_handlers) {
                struct early_suspend *e;
                e = list_entry(pos, struct early_suspend, link);
                if (e->level > handler->level)
                        break;
        }
        list_add_tail(&handler->link, pos);
        if ((state & SUSPENDED) && handler->suspend)
                handler->suspend(handler);
        mutex_unlock(&early_suspend_lock);
}
EXPORT_SYMBOL(register_early_suspend);
```

（2）取消已经注册的 Early Suspend 驱动

接口函数 EXPORT_SYMBOL(unregister_early_suspend)的功能是取消已经注册的 Early Suspend 的驱动，具体实现代码如下所示。

```
void unregister_early_suspend(struct early_suspend *handler)
{
        mutex_lock(&early_suspend_lock);
        list_del(&handler->link);
        mutex_unlock(&early_suspend_lock);
}
EXPORT_SYMBOL(unregister_early_suspend);
```

10.4.3 文件 wakelock.c

在 Power Management 系统的内核层中，实现文件 kernel/power/wakelock.c 对 Kernel 提供了如下接口函数。

（1）EXPORT_SYMBOL(wake_unlock)

此接口函数的功能是释放 lock，函数 wake_unlock()的具体实现代码如下所示。

```
void wake_unlock(struct wake_lock *lock)
{
        int type;
        unsigned long irqflags;
        spin_lock_irqsave(&list_lock, irqflags);
        type = lock->flags & WAKE_LOCK_TYPE_MASK;
#ifdef CONFIG_WAKELOCK_STAT
        wake_unlock_stat_locked(lock, 0);
#endif
        if (debug_mask & DEBUG_WAKE_LOCK)
                pr_info("wake_unlock: %s\n", lock->name);
        lock->flags &= ~(WAKE_LOCK_ACTIVE | WAKE_LOCK_AUTO_EXPIRE);
        list_del(&lock->link);
        list_add(&lock->link, &inactive_locks);
        if (type == WAKE_LOCK_SUSPEND) {
                long has_lock = has_wake_lock_locked(type);
                if (has_lock > 0) {
                        if (debug_mask & DEBUG_EXPIRE)
                                pr_info("wake_unlock: %s, start expire timer, "
```

```
                                        "%ld\n", lock->name, has_lock);
                        mod_timer(&expire_timer, jiffies + has_lock);
                } else {
                        if (del_timer(&expire_timer))
                                if (debug_mask & DEBUG_EXPIRE)
                                        pr_info("wake_unlock: %s, stop expire "
                                                "timer\n", lock->name);
                        if (has_lock == 0)
                                queue_work(suspend_work_queue, &suspend_work);
                }
                if (lock == &main_wake_lock) {
                        if (debug_mask & DEBUG_SUSPEND)
                                print_active_locks(WAKE_LOCK_SUSPEND);
#ifdef CONFIG_WAKELOCK_STAT
                        update_sleep_wait_stats_locked(0);
#endif
                }
        }
        spin_unlock_irqrestore(&list_lock, irqflags);
}
EXPORT_SYMBOL(wake_unlock);
```

（2）EXPORT_SYMBOL(wake_lock)

此接口函数的功能是申请 lock，必须调用相应的 unlock 来释放 lock。函数 wake_lock()的具体实现代码如下所示。

```
void wake_lock(struct wake_lock *lock)
{
        wake_lock_internal(lock, 0, 0);
}
EXPORT_SYMBOL(wake_lock);
```

（3）static DEFINE_TIMER(expire_timer, expire_wake_locks, 0, 0)

此接口函数的功能是如果定时时间到，则加入到 suspend 队列中。函数 expire_wake_locks()的具体实现代码如下所示。

```
static void expire_wake_locks(unsigned long data)
{
        long has_lock;
        unsigned long irqflags;
        if (debug_mask & DEBUG_EXPIRE)
                pr_info("expire_wake_locks: start\n");
        spin_lock_irqsave(&list_lock, irqflags);
        if (debug_mask & DEBUG_SUSPEND)
                print_active_locks(WAKE_LOCK_SUSPEND);
        has_lock = has_wake_lock_locked(WAKE_LOCK_SUSPEND);
        if (debug_mask & DEBUG_EXPIRE)
                pr_info("expire_wake_locks: done, has_lock %ld\n", has_lock);
        if (has_lock == 0)
                queue_work(suspend_work_queue, &suspend_work);
        spin_unlock_irqrestore(&list_lock, irqflags);
}
```

10.4.4　文件 resume.c

在 Power Management 系统的内核层中，实现文件 drivers/base/power/resume.c 对 Kernel 提供了如下接口函数。

（1）EXPORT_SYMBOL_GPL(device_power_up)

此接口函数的功能是打开特殊的设备，函数 device_power_up()的具体实现代码如下所示。

```
void device_power_up(void)
{
	sysdev_resume();
	dpm_power_up();
}
EXPORT_SYMBOL_GPL(device_power_up);
```

（2）EXPORT_SYMBOL_GPL(device_resume)

此接口函数的功能是重新存储设备的状态，函数 device_resume()的具体实现代码如下所示。

```
void device_resume(void)
{
	down(&dpm_sem);
	dpm_resume();
	up(&dpm_sem);
}
EXPORT_SYMBOL_GPL(device_resume);
```

10.4.5　文件 suspend.c

在 Power Management 系统的内核层中，实现文件 drivers/base/power/suspend.c 对 Kernel 提供了如下接口函数。

（1）EXPORT_SYMBOL_GPL(device_suspend)

此接口函数的功能是保存系统状态，并结束系统中进行的设备。函数 device_suspend()的具体实现代码如下所示。

```
int device_suspend(pm_message_t state)
{
	int error = 0;
	down(&dpm_sem);
	down(&dpm_list_sem);
	while (!list_empty(&dpm_active) && error == 0) {
		struct list_head * entry = dpm_active.prev;
		struct device * dev = to_device(entry);
		get_device(dev);
		up(&dpm_list_sem);
		error = suspend_device(dev, state);
		down(&dpm_list_sem);
		/* Check if the device got removed */
		if (!list_empty(&dev->power.entry)) {
			/* Move it to the dpm_off or dpm_off_irq list */
			if (!error) {
				list_del(&dev->power.entry);
```

```
				list_add(&dev->power.entry, &dpm_off);
			} else if (error == -EAGAIN) {
				list_del(&dev->power.entry);
				list_add(&dev->power.entry, &dpm_off_irq);
				error = 0;
			}
		}
		if (error)
			printk(KERN_ERR "Could not suspend device %s: "
				"error %d\n", kobject_name(&dev->kobj), error);
		put_device(dev);
	}
	up(&dpm_list_sem);
	if (error)
		dpm_resume();
	up(&dpm_sem);
	return error;
}
EXPORT_SYMBOL_GPL(device_suspend);
```

（2）EXPORT_SYMBOL_GPL(device_power_down)

此接口函数的功能是关闭特殊设备，函数 device_power_down()的具体实现代码如下所示。

```
int device_power_down(pm_message_t state)
{
	int error = 0;
	struct device * dev;
	list_for_each_entry_reverse(dev, &dpm_off_irq, power.entry) {
		if ((error = suspend_device(dev, state)))
			break;
	}
	if (error)
		goto Error;
	if ((error = sysdev_suspend(state)))
		goto Error;
 Done:
	return error;
 Error:
	dpm_power_up();
	goto Done;
}
EXPORT_SYMBOL_GPL(device_power_down);
```

10.4.6 文件 main.c

在 Power Management 系统的内核层中，内核文件 kernel/power/main.c 的主要实现代码如下所示。

```
static int __init pm_init(void)
{
	int error = pm_start_workqueue();
	if (error)
		return error;
	hibernate_image_size_init();
```

```
        hibernate_reserved_size_init();
        power_kobj = kobject_create_and_add("power", NULL);
        if (!power_kobj)
                return -ENOMEM;
        return sysfs_create_group(power_kobj, &attr_group);
}
core_initcall(pm_init);
```

在上述代码中，函数 pm_init(void)的返回值为 sysfs_create_group(power_kobj, &attr_group)，表示当对 sysfs/目录下相对的节点进行操作时会调用 attr_group 中的相关函数。

10.4.7　proc 文件

在 Power Management 系统的内核层中，给 Framework 层提供了如下的 proc 文件。

☑　/sys/android_power/acquire_partial_wake_lock：申请 partial wake lock。

☑　/sys/android_power/acquire_full_wake_lock：申请 full wakelock。

☑　/sys/android_power/release_wake_lock：释放相应的 wake lock。

☑　/sys/android_power/request_state：请求改变系统状态，有进入 standby 和回到 wakeup 两种状态。

☑　/sys/android_power/state：指示当前系统的状态。

Android 电源管理系统的主要功能是通过 Wake lock 来实现的，在最底层主要是通过如下 3 个队列来实现其管理功能。

☑　static LIST_HEAD(g_inactive_locks)

☑　static LIST_HEAD(g_active_partial_wake_locks)

☑　static LIST_HEAD(g_active_full_wake_locks)

在处理过程中实现如下插入和移动操作。

☑　所有被初始化的 lock 都会被插入到队列 g_inactive_locks 中。

☑　当前活动的 Partial Wake Lock 被插入到 g_active_partial_wake_locks 队列中。

☑　活动的 Full Wake Lock 被插入到 g_active_full_wake_locks 队列中。

☑　所有的 Partial Wake Lock 和 Full Wake Lock，在过期后或 unlock 后都会被移到 inactive 队列，以等待下次被调用。

10.5　wakelock 和 early_suspend

在 Android 系统中，wakelock 和 early_suspend 是一种特殊机制，能够实现系统的“唤醒”和“休眠”功能，获取系统资源的信息，例如电源信息和 CPU 信息等。本节将详细讲解 wakelock 和 early_suspend 机制的基本知识。

10.5.1　wakelock 的原理

wakelock 在 Android 的电源管理系统中扮演一个核心的角色。wakelock 是一种“锁”机制，只要有人拿着这个锁，系统就无法进入休眠状态。这个锁可以是有超时的或者是没有超时的，超时的锁会在时间过去以后自动解锁。如果没有锁了或者超时了，内核就会启动休眠的那套机制进入休眠。

当系统在启动完毕后，会自己去加一个名为 main 的锁，而当系统有意愿去睡眠时则会先释放这个 main

锁。在 Android 系统中，在 early_suspend 的最后一步会去释放 main 锁（wake_unlock: main）。释放完后则会去检查是否还有其他存在的锁，如果没有则直接进入睡眠过程。

它的缺点是，如果有某一应用获锁而不释放或者因一直在执行某种操作而没有时间来释放的话，则会导致系统一直进入不了睡眠状态，功耗过大。

在 wakelock 中有 3 种类型，最常用的是 WAKE_LOCK_SUSPEND，作用是防止系统进入睡眠。wakelock 的接口定义在文件 wakelock.c 中，定义代码如下所示。

```
enum {
    WAKE_LOCK_SUSPEND, /* Prevent suspend */
    WAKE_LOCK_IDLE, /* Prevent low power idle */
    WAKE_LOCK_TYPE_COUNT
};
```

在 wakelock 中，有如下两个地方可以让系统从 early_suspend 进入 suspend 状态。

- ☑ 在 wake_unlock 中，当解锁时，没有其他的 wakelock，则进入 suspend。
- ☑ 当超时锁的定时器超时后，定时器的回调函数会判断有没有其他的 wakelock，若没有则进入 suspend。

在 Android 系统中，在 Kernel 层使用 wakelock 的基本步骤如下。

（1）调用函数 android_init_suspend_lock()初始化一个 wakelock。

（2）调用相关申请 lock 的函数 android_lock_suspend()或 android_lock_suspend_auto_expire()请求 lock，此处只能申请 Partial Wake Lock，如果要申请 Full Wake Lock，则要调用函数 android_lock_partial_suspend_auto_expire()（该函数没有 EXPORT 出来）。

（3）如果是 auto expire 的 wakelock 则可忽略，否则必须及时把相关的 wakelock 释放掉，因为会造成系统长期运行在高功耗的状态。

（4）在驱动卸载或不再使用 wakelock 时，需要及时调用 android_uninit_suspend_lock 以释放资源。

由此可以总结出，Kernel 的 wakelock 唤醒操作的基本顺序依次是：

- ☑ 框架层函数的 acquireWakeLock()，此函数的具体实现代码如下所示。

```
int acquire_wake_lock(int lock, const char* id)
{
initialize_fds();
// LOGI("acquire_wake_lock lock=%d id='%s'\n", lock, id);
if (g_error) return g_error;
int fd;
if (lock == PARTIAL_WAKE_LOCK) {
fd = g_fds[ACQUIRE_PARTIAL_WAKE_LOCK];
}
else {
return EINVAL;
}
return write(fd, id, strlen(id));
}
```

- ☑ android_os_Power.cpp 的 acquireWakeLock()。
- ☑ power.c 的 acquire_wake_lock()。

10.5.2 early_suspend 原理

early_suspend 在 Linux 内核的睡眠过程前被调用。因为背光需要的能耗过大，所以常采用此类方法在手

机系统的设计中操作背光。如一些在内核中预先进行处理的事件可以先注册上 early_suspend()函数，这样当系统要进入睡眠之前会首先调用这些注册的函数。

和 Android 休眠唤醒相关的实现文件如下所示。

- ☑ linux_source/kernel/power/main.c
- ☑ linux_source/kernel/power/earlysuspend.c
- ☑ linux_source/kernel/power/wakelock.c
- ☑ linux_source/kernel/power/process.c
- ☑ linux_source/driver/base/power/main.c
- ☑ linux_source/arch/xxx/mach-xxx/pm.c 或 linux_source/arch/xxx/plat-xxx/pm.c

10.5.3　Android 休眠

当用户读写/sys/power/state 时，文件 linux_source/kernel/power/main.c 中的 state_store()函数会被调用。其中，Android 的 early_suspend 会执行：

```
request_suspend_state(state);
```

标准的 Linux 休眠会执行：

```
error = enter_state(state);
```

函数 state_store()的原型如下所示。

```
static ssize_t state_store(struct kobject *kobj, struct kobj_attribute *attr,
                        const char *buf, size_t n)
```

在函数 request_suspend_state()中，会调用 early_suspend_work 的工作队列以进入 early_suspend()函数中。函数 request_suspend_state()的原型如下所示。

```
void request_suspend_state(suspend_state_t new_state)
```

在函数 early_suspend()中，首先要判断当前请求的状态是否还是 suspend，如果不是则直接退出；如果是，则函数会调用已经注册的 early_suspend()函数。然后同步文件系统，最后释放 main_wake_lock。函数 early_suspend()的原型如下所示。

```
static void early_suspend(struct work_struct *work)
```

在函数 wake_unlock()中删除链表中的 wake_lock 节点，目的是判断当前是否存在 wake_lock。如果 wake_lock 的数目为 0，则调用工作队列 suspend_work，然后进入 suspend 状态。函数 wake_unlock()的原型如下所示。

```
void wake_unlock(struct wake_lock *lock)
```

在函数 suspend()中，首先判断当前是否有 wake_lock，如果有则退出，然后同步文件系统，最后调用 pm_suspend()函数。函数 suspend()的原型如下所示。

```
static void suspend(struct work_struct *work)
```

在函数 pm_suspend()中调用 enter_state()函数，这样就进入了标准 Linux 的休眠过程。函数 pm_ suspend()的原型如下所示。

```
int pm_suspend(suspend_state_t state)
```

在函数 enter_state()中，首先检查一些状态参数，再同步文件系统，然后调用 suspend_prepare()来冻结进程，最后调用 suspend_devices_and_enter()让外设进入休眠。函数 enter_state()的原型如下所示。

```
static int enter_state(suspend_state_t state)
```

在函数 suspend_prepare()中，先通过"pm_prepare_console();"给 suspend 分配一个虚拟终端来输出信息，再广播一个系统进入 suspend 的通报，关闭用户态的 helper 进程，然后调用函数 suspend_freeze_ processes()冻结进程，最后会尝试释放一些内存。函数 suspend_prepare()的原型如下所示。

```
static int suspend_prepare(void)
```

在函数 suspend_freeze_processes()中调用 freeze_processes()函数，而在 freeze_processes()函数中又调用了 try_to_freeze_tasks()函数来完成冻结任务。在冻结过程中，会判断当前进程是否有 wake_lock，如果有则冻结失败，函数会放弃冻结。函数 freeze_processes()的原型如下所示。

```
static int try_to_freeze_tasks(bool sig_only)
```

到此为止，所有的进程都已经停止了，内核态进程有可能在停止时还有一些信号量，如果这时在外设中去解锁这个信号量有可能会发生死锁，所以建议不要在外设的 suspend()中等待锁。而且在 suspend 的过程中，很多 log 是无法输出的，所以一旦出现问题就非常难以调试。

接下来回到 enter_state()函数中，当冻结进程完成后调用 suspend_devices_and_enter()函数，目的是让外设进入休眠。在该函数中，首先休眠串口，然后通过 device_suspend()函数调用各驱动的 suspend()函数。

当外设进入休眠后调用 suspend_ops->prepare()，suspend_ops 是板级的 PM 操作，假如是 s3c6410，则被注册在文件 linux_source/arch/arm/plat-s3c64xx/pm.c 中，在里面只定义了 suspend_ops->enter()函数。

```
static struct platform_suspend_ops s3c6410_pm_ops = {
	.enter		= s3c6410_pm_enter,
	.valid		= suspend_valid_only_mem,
};
```

然后在多 CPU 中关闭非启动 CPU，具体代码如下所示。

```
int suspend_devices_and_enter(suspend_state_t state)
{
	int error;
	if (!suspend_ops)
		return -ENOSYS;
	if (suspend_ops->begin) {
		error = suspend_ops->begin(state);
		if (error)
			goto Close;
	}
	suspend_console();
	suspend_test_start();
	error = device_suspend(PMSG_SUSPEND);
	if (error) {
		printk(KERN_ERR "PM: Some devices failed to suspend\n");
		goto Recover_platform;
	}
	suspend_test_finish("suspend devices");
	if (suspend_test(TEST_DEVICES))
		goto Recover_platform;
	if (suspend_ops->prepare) {
		error = suspend_ops->prepare();
		if (error)
			goto Resume_devices;
	}
	if (suspend_test(TEST_PLATFORM))
		goto Finish;
	error = disable_nonboot_cpus();
	if (!error && !suspend_test(TEST_CPUS))
		suspend_enter(state);
	enable_nonboot_cpus();
 Finish:
```

```
        if (suspend_ops->finish)
                suspend_ops->finish();
Resume_devices:
        suspend_test_start();
        device_resume(PMSG_RESUME);
        suspend_test_finish("resume devices");
        resume_console();
Close:
        if (suspend_ops->end)
                suspend_ops->end();
        return error;
Recover_platform:
        if (suspend_ops->recover)
                suspend_ops->recover();
        goto Resume_devices;
}
```

接下来调用函数 suspend_enter()，该函数将首先关闭 IRQ，然后调用 device_power_down()，此函数会调用 suspend_late()函数。这个函数是系统真正进入休眠最后调用的函数，通常会在这个函数中做最后的检查，接下来休眠所有的系统设备和总线。最后调用 suspend_pos->enter()使 CPU 进入省电状态。此时整个休眠过程完成了。函数 suspend_enter()的原型如下所示。

```
static int suspend_enter(suspend_state_t state)
```

10.5.4　Android 唤醒

如果在休眠中系统被中断或者其他事件唤醒，接下来的代码就从 suspend 完成的地方开始执行，以 s3c6410 为例，即从文件 pm.c 中的 s3c6410_pm_enter()中的 cpu_init()开始，然后执行 suspend_enter()的 sysdev_resume()函数，唤醒系统设备和总线，使能系统中断。然后回到 suspend_devices_and_enter()函数中，使能休眠时停止掉的非启动 CPU，并且继续唤醒每个设备。当函数 suspend_devices_and_enter()被执行完成后，系统外设已经唤醒，但进程依然处于冻结的状态，返回到 enter_state 函数中，调用 suspend_finish()函数。在函数 suspend_finish()中解冻进程和任务，这样可以使用户空间帮助进程，从而广播一个系统从 suspend 状态退出的 notify。

当所有的唤醒已经结束以后，用户进程都已经开始运行了，但没有点亮屏幕，唤醒通常会是以下几种原因。

（1）如果是来电，那么 Modem 会发送命令给 rild，这样可以让 rild 通知 WindowManager 有来电响应，这样就会远程调用 PowerManagerService 来写 on 到/sys/power/state 来调用 late resume()，执行点亮屏幕等操作。

（2）用户按键事件会送到 WindowManager 中，WindowManager 会处理这些按键事件，按键分为几种情况，如果按键不是唤醒键，那么 WindowManager 会主动放弃 wakelock 使系统进入再次休眠；如果按键是唤醒键，那么 WindowManager 就会调用 PowerManagerService 中的接口来执行 late Resume。

（3）当 on 被写入/sys/power/state 之后，同 early_suspend 过程一样，request_suspend_state()会被调用，只是执行的工作队列变为 late_resume_work。在 late_resume()函数中，唤醒调用了 early_suspend 的设备。

10.6　Battery 电池系统架构和管理

Android 中的电池使用方式主要有 3 种不同的模式，分别是 AC、USB 和 Battery。因为在应用程序层次

中通常包括了电池状态显示的功能，所以从 Android 系统的软件方面（包括驱动程序和用户空间内容）需要在一定程度上获得电池的状态，电池系统主要负责电池信息统计、显示。Android 电池系统的架构如图 10-3 所示。

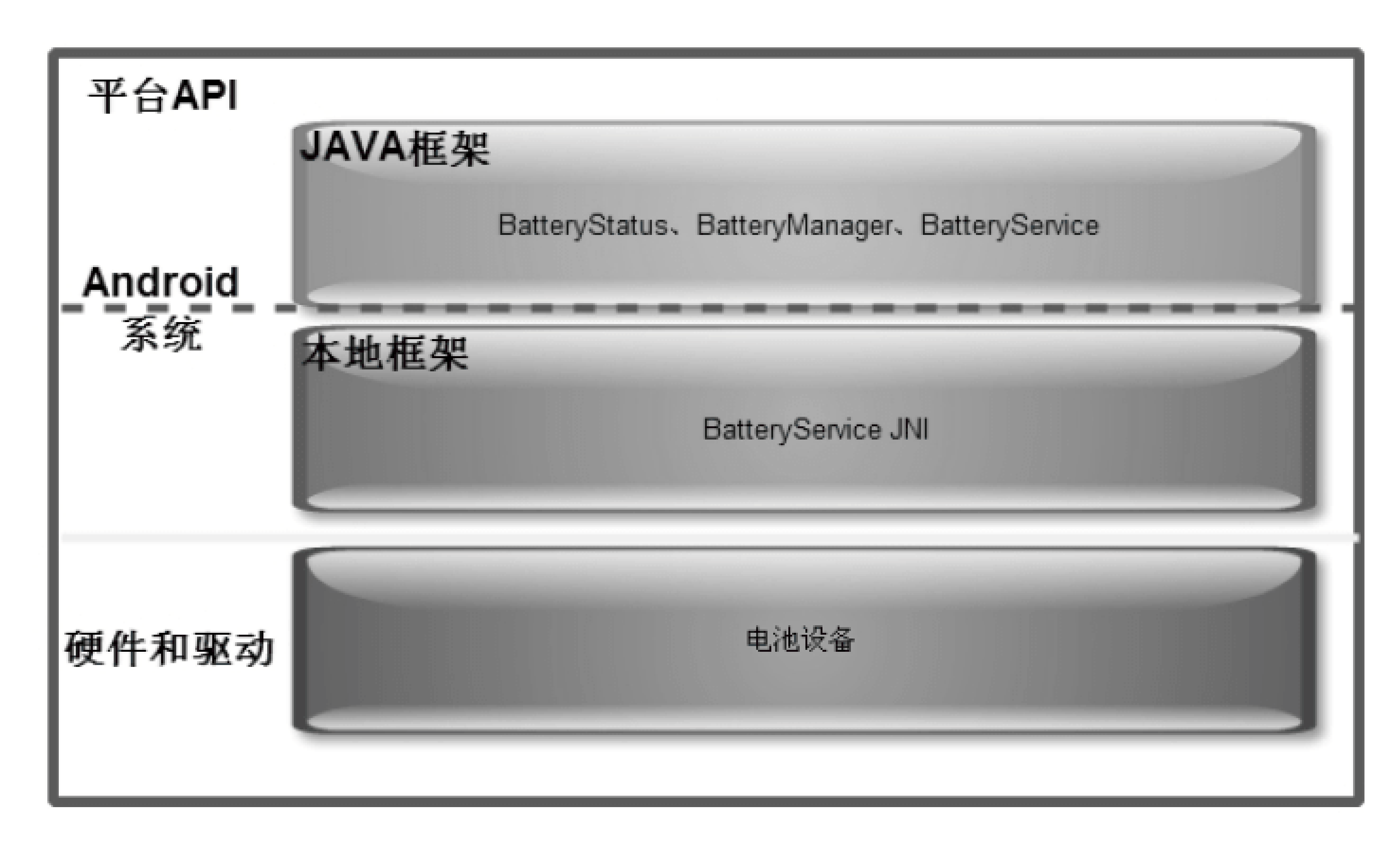


图 10-3　Android 电池系统的架构图

自下而上，Android 的电池系统分成以下所示的几个部分。

10.6.1　实现驱动程序

特定硬件平台电池的驱动程序，用 Linux 的 Power Supply 驱动程序，实现向用户空间提供信息。Battery 驱动程序需要通过 sys 文件系统向用户空间提供接口，sys 文件系统的路径是由上层的程序指定的。Linux 标准的 Power Supply 驱动程序所使用的文件系统路径为/sys/class/power_supply，其中的每个子目录表示一种能源供应设备的名称，如图 10-4 所示。

Power Supply 驱动程序的头文件在 include/linux/power_supply.h 中定义，实现注册和注销驱动程序的函数如下所示。

```
int power_supply_register(struct device *parent,struct power_supply *psy);
void power_supply_unregister(struct power_supply *psy);
struct power_supply {
const char *name;
/*  设备名称  */
enum power_supply_type type;
/*  类型  */
enum power_supply_property *properties;
/*  属性指针  */
size_t num_properties;
/*  属性的数目  */
char **supplied_to;
size_t num_supplicants;
int (*get_property)(struct power_supply *psy, /*  获得属性  */
enum power_supply_property psp,
union power_supply_propval *val);
void (*external_power_changed)(struct power_supply *psy);
/* ...省略部分内容  */
};
```

Linux 中的驱动程序为 power_supply，如图 10-5 所示。

```
/sys/class/power_supply # pwd
/sys/class/power_supply
/sys/class/power_supply # ls
battery
usb
ac
/sys/class/power_supply #
```

图 10-4　3 种不同的模式

```
tim@tim-Inspiron-N5110:~/android_4412_svn_03/kernel/drivers/power$ ls power_supply
power_supply_core.c  power_supply_core.o  power_supply.h  power_supply_leds.c  pow
tim@tim-Inspiron-N5110:~/android_4412_svn_03/kernel/drivers/power$ ll power_supply
-rw-rw-r-- 1 tim tim  6280 2012-12-18 14:29 power_supply_core.c
-rw-rw-r-- 1 tim tim 46472 2013-01-11 17:41 power_supply_core.o
-rw-rw-r-- 1 tim tim  1117 2012-12-18 14:29 power_supply.h
-rw-rw-r-- 1 tim tim  4878 2012-12-18 14:29 power_supply_leds.c
-rw-rw-r-- 1 tim tim 83187 2013-01-11 17:41 power_supply.o
-rw-rw-r-- 1 tim tim  8049 2012-12-18 14:29 power_supply_sysfs.c
-rw-rw-r-- 1 tim tim 38536 2013-01-11 17:41 power_supply_sysfs.o
tim@tim-Inspiron-N5110:~/android_4412_svn_03/kernel/drivers/power$
```

图 10-5　Linux 中的驱动程序 power_supply

10.6.2　实现 JNI 本地代码

Android 电池系统的代码路径为 frameworks/base/services/jni/com_android_server_BatteryService.cpp。

类 com_android_server_BatteryService 调用 sys 文件系统访问驱动程序，也同时提供了 JNI 的接口，这个文件提供的方法列表如下所示。

```
static JNINativeMethod sMethods[] = {
{"native_update", "()V", (void*)android_server_BatteryService_update},
};
```

本地 JNI 的处理流程如下。

（1）根据设备类型判定设备后，得到各个设备的相关属性。

（2）如果是交流或者 USB 设备，则只需要得到它们是否在线（onLine）。

（3）如果是电池设备，则需要得到更多的信息。例如状态（status）、健康程度（health）、容量（capacity）和电压（voltage_now）等。

Linux 驱动 Driver 维护着保存电池信息的一组文件 sysfs，功能是供应用程序获取电源的相关状态，具体说明如下。

- ☑ #define AC_ONLINE_PATH "/sys/class/power_supply/ac/online" AC 电源连接状态。
- ☑ #define USB_ONLINE_PATH "/sys/class/power_supply/usb/online" USB 电源连接状态。
- ☑ #define BATTERY_STATUS_PATH "/sys/class/power_supply/battery/status"充电状态。
- ☑ #define BATTERY_HEALTH_PATH "/sys/class/power_supply/battery/health"电池状态。
- ☑ #define BATTERY_PRESENT_PATH "/sys/class/power_supply/battery/present"使用状态。
- ☑ #define BATTERY_CAPACITY_PATH "/sys/class/power_supply/battery/capacity"电池 level。
- ☑ #define BATTERY_VOLTAGE_PATH "/sys/class/power_supply/battery/batt_vol"电池电压。
- ☑ #define BATTERY_TEMPERATURE_PATH "/sys/class/power_supply/battery/batt_temp"电池温度。
- ☑ #define BATTERY_TECHNOLOGY_PATH "/sys/class/power_supply/battery/technology"电池技术，当电池状态发生变化时，driver 会更新这些文件传送信息到 Java 层。

10.6.3　Java 层代码

Android 电池系统的 Java 层代码路径如下。

- ☑ frameworks/base/services/java/com/android/server/BatteryService.java：电池服务文件。
- ☑ frameworks/base/core/java/android/os/：android.os 包中和 Battery 相关的部分。
- ☑ frameworks/base/core/java/com/android/internal/os/：和 Battery 相关的内部部分 BatteryService.java 通过调用 BatteryService JNI 来实现 com.android.server 包中的 BatteryService 类。BatteryManager.java

中定义了一些 Java 应用程序层可以使用的常量。

上述文件的调用运行关系如图 10-6 所示。

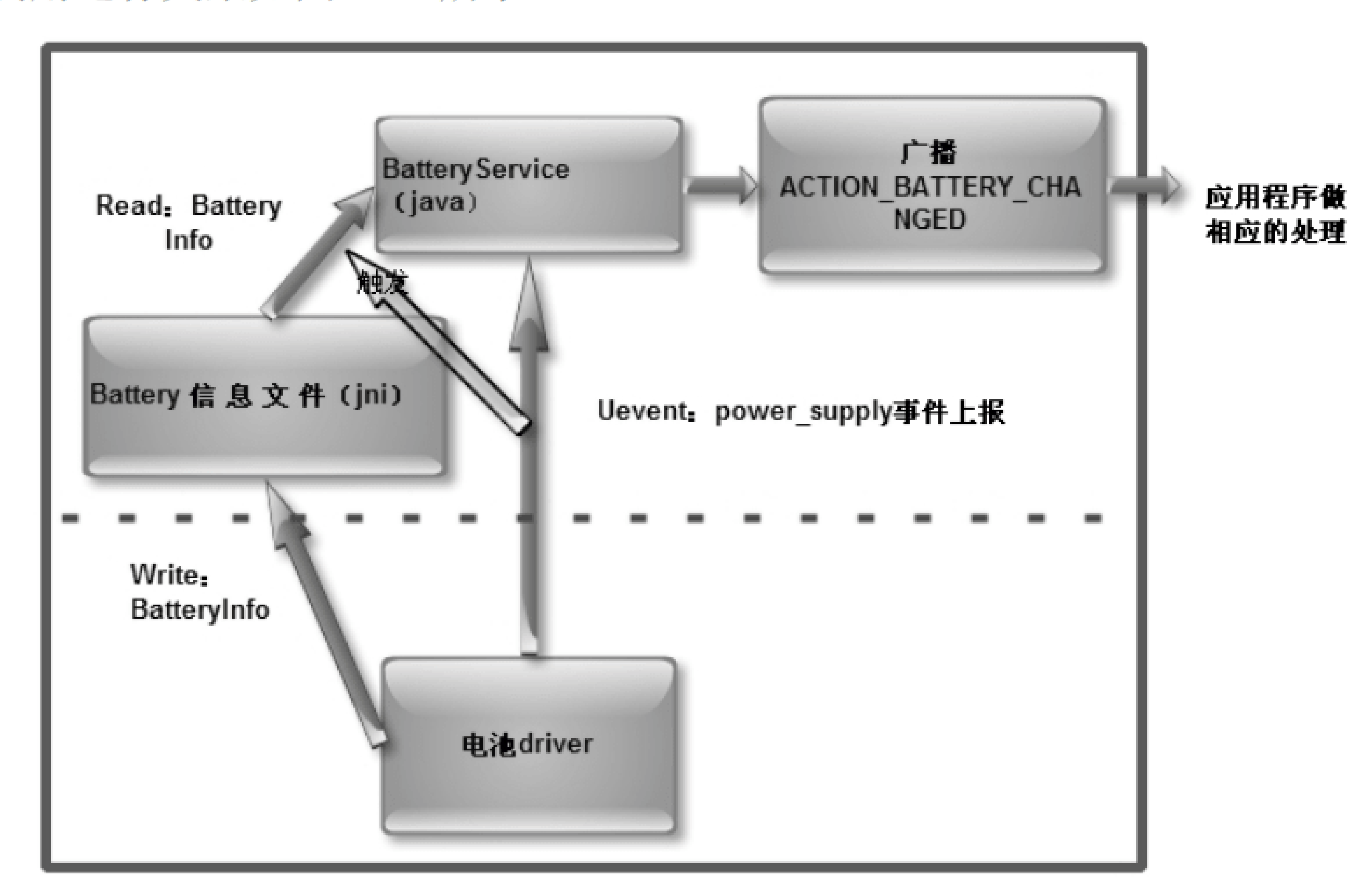


图 10-6　Java 层的调用运行关系

Android 电池系统在驱动程序层以上的部分都是 Android 系统中默认的内容。在移植的过程中基本不需要改动，电池系统需要移植的部分仅有 Battery 驱动程序。Battery 驱动程序使用 Linux 标准的 Power Supply 驱动程序与上层进行通信，通信的接口是 sys 文件系统，通过这个接口能读取 sys 文件系统中的文件来获取电池相关的信息。

BatteryService 作为电池及充电相关的服务，功能是监听 UEvent、读取 sysfs 中的状态、广播信息 Intent. ACTION_BATTERY_CHANGED。BatteryService 实现了一个 UEventObserver mUEventObserver。uevent 是 Linux 内核用来向用户空间主动上报事件的机制，对于 Java 程序来说，只实现 UEventObserver 的虚函数 onUEvent，然后注册即可。BatteryService 只关注 power_supply 的事件，所以在构造函数中实现注册。在文件 BatteryService.java 中，实现 UEventObserver 的代码如下所示。

```
private final UEventObserver mInvalidChargerObserver = new UEventObserver() {
    @Override
    public void onUEvent(UEventObserver.UEvent event) {
        final int invalidCharger = "1".equals(event.get("SWITCH_STATE")) ? 1 : 0;
        synchronized (mLock) {
            if (mInvalidCharger != invalidCharger) {
                mInvalidCharger = invalidCharger;
            }
        }
    }
};
```

在文件 BatteryService.java 中，函数 update()用于读取 sysfs 文件做到同步取得电池信息，然后根据读到的状态更新 BatteryService 的成员变量，并广播一个 Intent 来通知其他关注电源状态的组件。

当 Kernel 有 power_supply 事件上报时，mUEventObserver 调用 update()函数，然后 update 调用 native_ update 从 sysfs 中读取相关状态(com_android_server_BatteryService.cpp)。update()函数的具体实现代码如下所示。

```
private void update(BatteryProperties props) {
    synchronized (mLock) {
        if (!mUpdatesStopped) {
```

```
                mBatteryProps = props;
                // Process the new values.
                processValuesLocked();
            }
        }
}
```

10.6.4　实现 Uevent 部分

Uevent 是内核通知 Android 有状态变化的一种方法，例如 USB 线插入、拔出，电池电量变化等。其本质是内核发送（可以通过 socket）一个字符串，应用层（Android）接收并解释该字符串，获取相应信息。如图 10-7 所示，如果其中有信息变化，Uevent 触发，做出相应的数据更新。

```
root@android:/sys/class/power_supply/ac # cd ../
root@android:/sys/class/power_supply # ls
ac
battery
root@android:/sys/class/power_supply # cd battery/
root@android:/sys/class/power_supply/battery # ls
capacity
device
health
power
present
status
subsystem
technology
temp
type
uevent
voltage_now
root@android:/sys/class/power_supply/battery # cat capacity
90
root@android:/sys/class/power_supply/battery # cat health
Good
root@android:/sys/class/power_supply/battery #
```

图 10-7　Uevent 获取信息

Android 中的 BatteryService 及相关组件如下。

（1）Androiduevent 架构

Android 系统中的很多事件都是通过 UEvent 和 kernel 来异步通信的，其中类 UEventObserver 是核心。UEventObserver 接收 kernel 的 uevent 信息的抽象类。

Server 层代码的实现文件如下。

☑ frameworks/frameworks/base/services/java/com/android/server/SystemServer.java
☑ frameworks/frameworks/base/services/java/com/android/server/BatteryService.java

Java 层代码的实现文件是：frameworks/base/core/java/android/os/UEventObserver.java。

JNI 层代码的实现文件是：frameworks/base/core/jni/android_os_UEventObserver.cpp。

底层代码的实现文件是：hardware/libhardware_legacy/uevent/uevent.c。

读写 Kernel 的接口是：socket(PF_NETLINK,SOCK_DGRAM, NETLINK_KOBJECT_UEVENT)。

（2）UEventObserver 类的使用

在文件 UEventObserver.java 中，提供了如下 3 个接口供子类进行调用。

☑ onUEvent(UEvent event)：子类必须重写这个 onUEvent 来处理 UEvent，具体实现代码如下所示。

```
public static final class UEvent {
    // collection of key=value pairs parsed from the uevent message
    private final HashMap<String,String> mMap = new HashMap<String,String>();
```

```
    public UEvent(String message) {
        int offset = 0;
        int length = message.length();

        while (offset < length) {
            int equals = message.indexOf('=', offset);
            int at = message.indexOf('\0', offset);
            if (at < 0) break;

            if (equals > offset && equals < at) {
                // key is before the equals sign, and value is after
                mMap.put(message.substring(offset, equals),
                        message.substring(equals + 1, at));
            }

            offset = at + 1;
        }
    }

    public String get(String key) {
        return mMap.get(key);
    }

    public String get(String key, String defaultValue) {
        String result = mMap.get(key);
        return (result == null ? defaultValue : result);
    }

    public String toString() {
        return mMap.toString();
    }
}
```

☑ startObserving(Stringmatch)：启动进程，要提供一个字符串参数，具体实现代码如下所示。

```
public final void startObserving(String match) {
    if (match == null || match.isEmpty()) {
        throw new IllegalArgumentException("match substring must be non-empty");
    }

    final UEventThread t = getThread();
    t.addObserver(match, this);
}
```

☑ stopObserving()：停止进程，具体实现代码如下所示。

```
public final void stopObserving() {
    final UEventThread t = getThread();
    if (t != null) {
        t.removeObserver(this);
    }
}
```

在 UEvent thread 中会不停调用 update()方法，来更新电池的信息数据。

（3）vold server 分析

在文件 system/vold/NetlinkManager.cpp 中的实现代码如下所示。

```
if ((mSock = socket(PF_NETLINK,SOCK_DGRAM,NETLINK_KOBJECT_UEVENT)) < 0) {
                SLOGE("Unable to create uevent socket: %s", strerror(errno));
                return -1;
        }
        if (setsockopt(mSock, SOL_SOCKET, SO_RCVBUFFORCE, &sz, sizeof(sz)) < 0) {
                SLOGE("Unable to set uevent socket options: %s", strerror(errno));
                return -1;
        }
        if (bind(mSock, (struct sockaddr *) &nladdr, sizeof(nladdr)) < 0) {
                SLOGE("Unable to bind uevent socket: %s", strerror(errno));
                return -1;
        }
```

在文件 system/vold/NetlinkHandler.cpp 的 NetlinkHandler::onEvent 中的处理代码如下所示。

```
void NetlinkHandler::onEvent(NetlinkEvent *evt) {
    VolumeManager *vm = VolumeManager::Instance();
    const char *subsys = evt->getSubsystem();
    if (!subsys) {
        SLOGW("No subsystem found in netlink event");
        return;
    }
    if (!strcmp(subsys, "block")) {
        vm->handleBlockEvent(evt);
    } else if (!strcmp(subsys, "switch")) {
        vm->handleSwitchEvent(evt);
    } else if (!strcmp(subsys, "battery")) {
    } else if (!strcmp(subsys, "power_supply")) {
    }
}
```

最后在文件 system/core/libsysutils/src/NetlinkListener.cpp 中实现监听。

（4）Batteryserver 分析

Java 层的实现代码位于 frameworks/frameworks/base/services/java/com/android/server/BatteryService.java。

NI 层的实现代码位于 frameworks/base/services/jni/com_android_server_BatteryService.cpp。

BatteryService 运行在 system_process 中，在系统初始化时启动，例如在文件 BatteryService.java 中的对应代码如下所示。

```
Log.i(TAG, "Starting Battery Service.");
BatteryService battery = new BatteryService(context);
ServiceManager.addService("battery", battery);
```

BatteryService 通过 JNI（com_android_server_BatteryService.cpp）来读取数据。BatteryService 通过 JNI 不仅注册函数而且还注册变量。BatteryService 运行在 system_process 中，在系统初始化时启动，例如在文件 BatteryService.java 中声明变量的代码如下所示。

```
//##############在 BatteryService.java 中声明的变量################
private boolean mAcOnline;
private boolean mUsbOnline;
private int mBatteryStatus;
private int mBatteryHealth;
private boolean mBatteryPresent;
```

```
private int mBatteryLevel;
private int mBatteryVoltage;
private int mBatteryTemperature;
private String mBatteryTechnology;
//在 BatteryService.java 中声明的变量，在 com_android_server_BatteryService.cpp 中共用，即在 com_android_server_BatteryService.cpp 中其实操作的也是 BatteryService.java 中声明的变量
gFieldIds.mAcOnline = env->GetFieldID(clazz, "mAcOnline", "Z");
gFieldIds.mUsbOnline = env->GetFieldID(clazz, "mUsbOnline", "Z");
gFieldIds.mBatteryStatus = env->GetFieldID(clazz, "mBatteryStatus", "I");
gFieldIds.mBatteryHealth = env->GetFieldID(clazz, "mBatteryHealth", "I");
gFieldIds.mBatteryPresent = env->GetFieldID(clazz, "mBatteryPresent", "Z");
gFieldIds.mBatteryLevel = env->GetFieldID(clazz, "mBatteryLevel", "I");
gFieldIds.mBatteryTechnology = env->GetFieldID(clazz, "mBatteryTechnology", "Ljava/lang/String; ");
gFieldIds.mBatteryVoltage = env->GetFieldID(clazz, "mBatteryVoltage", "I");
gFieldIds.mBatteryTemperature = env->GetFieldID(clazz, "mBatteryTemperature", "I");
//上面这些变量的值，对应是从下面的文件中读取的，一个文件存储一个数值
#define AC_ONLINE_PATH "/sys/class/power_supply/ac/online"
#define USB_ONLINE_PATH "/sys/class/power_supply/usb/online"
#define BATTERY_STATUS_PATH "/sys/class/power_supply/battery/status"
#define BATTERY_HEALTH_PATH "/sys/class/power_supply/battery/health"
#define BATTERY_PRESENT_PATH "/sys/class/power_supply/battery/present"
#define BATTERY_CAPACITY_PATH "/sys/class/power_supply/battery/capacity"
#define BATTERY_VOLTAGE_PATH "/sys/class/power_supply/battery/batt_vol"
#define BATTERY_TEMPERATURE_PATH "/sys/class/power_supply/battery/batt_temp"
#define BATTERY_TECHNOLOGY_PATH "/sys/class/power_supply/battery/technology"
```

BatteryService 主动把数据传送给所关心的应用程序，所有的电池的信息数据是通过 Intent 传送出去的。在文件 BatteryService.java 中，实现数据传送功能的实现代码如下所示。

```
Intent intent = new Intent(Intent.ACTION_BATTERY_CHANGED);
intent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY);
intent.putExtra("status", mBatteryStatus);
intent.putExtra("health", mBatteryHealth);
intent.putExtra("present", mBatteryPresent);
intent.putExtra("level", mBatteryLevel);
intent.putExtra("scale", BATTERY_SCALE);
intent.putExtra("icon-small", icon);
intent.putExtra("plugged", mPlugType);
intent.putExtra("voltage", mBatteryVoltage);
intent.putExtra("temperature", mBatteryTemperature);
intent.putExtra("technology", mBatteryTechnology);
ActivityManagerNative.broadcastStickyIntent(intent, null);
```

应用程序如果想要接收到 BatteryService 发送出来的电池信息，则需要注册一个 Intent 为 Intent.ACTION_BATTERY_CHANGED 的 BroadcastReceiver。实现数据接收功能的注册方法如下所示。

```
IntentFilter mIntentFilter = new IntentFilter();
mIntentFilter.addAction(Intent.ACTION_BATTERY_CHANGED);
registerReceiver(mIntentReceiver, mIntentFilter);
private BroadcastReceiver mIntentReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub
        String action = intent.getAction();
```

```
            if (action.equals(Intent.ACTION_BATTERY_CHANGED)) {
                int nVoltage = intent.getIntExtra("voltage", 0);
                if(nVoltage!=0){
                    mVoltage.setText("V: " + nVoltage + "mV – Success…");
                }
                else{
                    mVoltage.setText("V: " + nVoltage + "mV – fail…");
                }
            }
        }
};
```

电池的信息会随着时间不停变化，自然地就需要考虑如何实时更新电池的数据信息。在 BatteryService 启动时，会同时通过 UEventObserver 启动一个 onUEvent Thread。每一个 Process 最多只能有一个 onUEvent Thread，即使这个 Process 中有多个 UEventObserver 的实例。当在一个 Process 中，第一次使用 Call startObserving()方法后，这个 UEvent thread 就启动了，而一旦这个 UEvent thread 启动之后就不会停止。在 BatteryService.java 中实现数据更新功能的代码如下所示。

```
mUEventObserver.startObserving("SUBSYSTEM=power_supply");
private UEventObserver mUEventObserver = new UEventObserver() {
    @Override
    public void onUEvent(UEventObserver.UEvent event) {
        update();
    }
};
```

在 UEvent thread 中会不停调用 update()方法来更新电池的信息数据。

10.7　JobScheduler 节能调度机制

耗电量大一直是 Android 智能设备需要解决的问题，由于系统原理的问题，这方面一直处理的不够好，大多数设备只能适度地使用一天，如果不充电很少可以连续使用两天。从 Android 5.0 版本开始，将通过 Volta 中的 JobScheduler 机制对这个问题作出一些改进，它通过改善第三方程序的工作序列来降低程序的耗电量。本节将详细讲解 JobScheduler 机制的基本知识。

10.7.1　JobScheduler 机制的推出背景

提高电池续航，也就意味着减少系统和程序的电量消耗。为此 Google 工程师们经过测试发现，每次唤醒设备 1～2 秒时，都会消耗 2 分钟的待机电量。由此可见，每次唤醒设备时，不仅仅是点亮了屏幕，系统也在后台处理很多事情。而 Android 5.0 版本为了解决这个问题，使用了一个新的 API JobScheduler，它可以让系统批处理一些不重要的 App 请求，例如数据库清理和日志上传等。研发人员也可以使用这个 API 减少自己 App 的不必要操作。

过去，如果开发人员想通过后台调取服务器数据或完成某些处理工作，应用程序必须先监听是否有事件正在发生，并为自己设定一个唤醒时间。这样每当应用程序开始运行时，开发人员需要检查各种环境条件，以确定是否具备条件让它完成工作还是需要稍后再试。上述检查方式不仅复杂而且容易出错，会不断地浪费资源。例如当一个应用程序被唤醒后，发现条件不符合就只能去睡觉并为下次唤醒再次设定时间，

这是一个反复的过程。

从 Android 5.0 版本开始，上述问题将引用 JobScheduler 机制来修复，它作为一个调度应用程序，负责当应用程序被唤醒时，提供适当的运行环境，所以开发者不用再让程序检测环境是否符合需求，开发人员只需要按照标准的流程，调度程序会自动为唤醒的程序准备好运行环境。

应用程序可以使用这个调度程序来唤醒它们，例如当设备连接到充电器后，调度程序将唤醒那些需要处理器工作的程序，让它们进行工作，或者在设备连接至 WiFi 网络时上传下载照片、更新内容等。该调度程序还支持一个时间窗口，以便它可以唤醒一组应用程序，这将使那些不需要精确唤醒时间，但每隔一两个小时需要运行一次的程序能在同一时间点运行，这样就能让处理器保持更长时间的休眠。

JobScheduler 的优势相当大，它不仅可以帮助手机节省电量，实际上由于不需要再监听、更改和设置报警，还可以帮助开发人员减少代码书写量。

10.7.2 JobScheduler 的实现

类 JobScheduler 的实现文件是 platform/frameworks/base/core/java/android/app/job/JobScheduler.java，这是从 Android 5.0 开始提供的一种全新 API，可以通过定义 Job 系统以异步方式在稍后的时间或在特定条件（例如当设备正在充电时）下运行，以达到优化电池寿命的目的。类 JobScheduler 的具体功能如下。

☑ 推迟非面向用户的工作。
☑ 设置有一个需要访问网络或 WiFi 连接的任务。
☑ 设置应用程序都有一个编号，可以作为一个批次上定期运行的任务。

文件 platform/frameworks/base/core/java/android/app/job/JobScheduler.java 的主要实现代码如下所示。

```
public abstract class JobScheduler {
    /**从 schedule(JobInfo) 返回的一个无效的参数*/
    public static final int RESULT_FAILURE = 0;
    /**
     从 schedule(JobInfo)返回的值，表示返回超时请求
     */
    public static final int RESULT_SUCCESS = 1;
    public abstract int schedule(JobInfo job);
    /**
     * 取消一个待解决的 jobscheduler 工作
     */
    public abstract void cancel(int jobId);
    /**
     * 取消所有已注册的 jobscheduler 包的工作
     */
    public abstract void cancelAll();
    /**
     *定义一个所有通过这种包还没有被执行登记工作的列表
     */
    public abstract List<JobInfo> getAllPendingJobs();
}
```

10.7.3 实现操作调度

在 Android 5.0 系统中，JobScheduler API 将需要做的事情全部由类 JobService 来控制。类 JobService 在

文件 platform/frameworks/base/core/java/android/app/job/JobService.java 中定义，JobService 作为一个大容器封装了 JobScheduler 传递的数据，这些数据能够设置对操控应用程序的工作调度参数。文件 JobService.java 的具体实现代码如下所示。

```
public abstract class JobService extends Service {
    private static final String TAG = "JobService";
    public static final String PERMISSION_BIND =
            "android.permission.BIND_JOB_SERVICE";
    private final int MSG_EXECUTE_JOB = 0;
    private final int MSG_STOP_JOB = 1;
    private final int MSG_JOB_FINISHED = 2;

    private final Object mHandlerLock = new Object();

    @GuardedBy("mHandlerLock")
    JobHandler mHandler;

    /** Binder for this service. */
    IJobService mBinder = new IJobService.Stub() {
        @Override
        public void startJob(JobParameters jobParams) {
            ensureHandler();
            Message m = Message.obtain(mHandler, MSG_EXECUTE_JOB, jobParams);
            m.sendToTarget();
        }
        @Override
        public void stopJob(JobParameters jobParams) {
            ensureHandler();
            Message m = Message.obtain(mHandler, MSG_STOP_JOB, jobParams);
            m.sendToTarget();
        }
    };

    /** @hide */
    void ensureHandler() {
        synchronized (mHandlerLock) {
            if (mHandler == null) {
                mHandler = new JobHandler(getMainLooper());
            }
        }
    }
    class JobHandler extends Handler {
        JobHandler(Looper looper) {
            super(looper);
        }

        @Override
        public void handleMessage(Message msg) {
            final JobParameters params = (JobParameters) msg.obj;
            switch (msg.what) {
                case MSG_EXECUTE_JOB:
```

```
                try {
                    boolean workOngoing = JobService.this.onStartJob(params);
                    ackStartMessage(params, workOngoing);
                } catch (Exception e) {
                    Log.e(TAG, "Error while executing job: " + params.getJobId());
                    throw new RuntimeException(e);
                }
                break;
            case MSG_STOP_JOB:
                try {
                    boolean ret = JobService.this.onStopJob(params);
                    ackStopMessage(params, ret);
                } catch (Exception e) {
                    Log.e(TAG, "Application unable to handle onStopJob.", e);
                    throw new RuntimeException(e);
                }
                break;
            case MSG_JOB_FINISHED:
                final boolean needsReschedule = (msg.arg2 == 1);
                IJobCallback callback = params.getCallback();
                if (callback != null) {
                    try {
                        callback.jobFinished(params.getJobId(), needsReschedule);
                    } catch (RemoteException e) {
                        Log.e(TAG, "Error reporting job finish to system: binder has gone" +
                                "away.");
                    }
                } else {
                    Log.e(TAG, "finishJob() called for a nonexistent job id.");
                }
                break;
            default:
                Log.e(TAG, "Unrecognised message received.");
                break;
        }
    }

    private void ackStartMessage(JobParameters params, boolean workOngoing) {
        final IJobCallback callback = params.getCallback();
        final int jobId = params.getJobId();
        if (callback != null) {
            try {
                 callback.acknowledgeStartMessage(jobId, workOngoing);
            } catch(RemoteException e) {
                Log.e(TAG, "System unreachable for starting job.");
            }
        } else {
            if (Log.isLoggable(TAG, Log.DEBUG)) {
                Log.d(TAG, "Attempting to ack a job that has already been processed.");
            }
        }
```

```
        }

        private void ackStopMessage(JobParameters params, boolean reschedule) {
            final IJobCallback callback = params.getCallback();
            final int jobId = params.getJobId();
            if (callback != null) {
                try {
                    callback.acknowledgeStopMessage(jobId, reschedule);
                } catch(RemoteException e) {
                    Log.e(TAG, "System unreachable for stopping job.");
                }
            } else {
                if (Log.isLoggable(TAG, Log.DEBUG)) {
                    Log.d(TAG, "Attempting to ack a job that has already been processed.");
                }
            }
        }
    }
    public abstract boolean onStartJob(JobParameters params);

    public abstract boolean onStopJob(JobParameters params);
    public final void jobFinished(JobParameters params, boolean needsReschedule) {
        ensureHandler();
        Message m = Message.obtain(mHandler, MSG_JOB_FINISHED, params);
        m.arg2 = needsReschedule ? 1 : 0;
        m.sendToTarget();
    }
}
```

由此可见，JobService 服务在执行每个在程序的主线程 Handler 中运行的工作时，意味着必须卸载执行逻辑，从另一个“线程/处理器/AsyncTask”进行选择。如果不这样做，会导致阻止从 JobManager 发出的任何回调任务。

10.7.4　封装调度任务

在 Android 5.0 系统中，类 JobInfo 功能封装全部的调度任务。类 JobInfo 在文件 platform/frameworks/base/core/java/android/app/job/JobInfo.java 中定义，定义调度参数的代码如下所示。

```
public class JobInfo implements Parcelable {
    /**默认值 */
    public static final int NETWORK_TYPE_NONE = 0;
    /** 需要网络连接的工作 */
    public static final int NETWORK_TYPE_ANY = 1;
    /**需要网络连接的工作，无须计量*/
    public static final int NETWORK_TYPE_UNMETERED = 2;

    /**
     *补偿工作已在默认情况下，以毫秒为单位
     */
    public static final long DEFAULT_INITIAL_BACKOFF_MILLIS = 30000L;   // 30 seconds.

    /**
```

```
    *可以工作时的最大回退值，单位为毫秒
    */
   public static final long MAX_BACKOFF_DELAY_MILLIS = 5 * 60 * 60 * 1000;   // 5 hours.

   /**
   线性回退失败的作业
    */
   public static final int BACKOFF_POLICY_LINEAR = 0;

   /**
    *指数回退失败的作业
    */
   public static final int BACKOFF_POLICY_EXPONENTIAL = 1;
   public static final int DEFAULT_BACKOFF_POLICY = BACKOFF_POLICY_EXPONENTIAL;

   private final int jobId;
   private final PersistableBundle extras;
   private final ComponentName service;
   private final boolean requireCharging;
   private final boolean requireDeviceIdle;
   private final boolean hasEarlyConstraint;
   private final boolean hasLateConstraint;
   private final int networkType;
   private final long minLatencyMillis;
   private final long maxExecutionDelayMillis;
   private final boolean isPeriodic;
   private final boolean isPersisted;
   private final long intervalMillis;
   private final long initialBackoffMillis;
   private final int backoffPolicy;
```

在文件 JobInfo.java 中，通过类 Builder 来配置应该运行的计划任务，期间可以安排执行任务的具体条件，例如在运行下列应用情形时：

☑ 当此设备开始充电时。

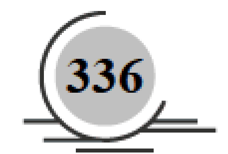
☑ 开始时，该装置被连接到一个未计量的网络。

☑ 开始时，该设备处于空闲状态。

☑ 往前一定期限，或以最小的延迟结束。

类 Builder 的具体实现代码如下所示。

```
   public static final class Builder {
       private int mJobId;
       private PersistableBundle mExtras = PersistableBundle.EMPTY;
       private ComponentName mJobService;
       // Requirements.
       private boolean mRequiresCharging;
       private boolean mRequiresDeviceIdle;
       private int mNetworkType;
       private boolean mIsPersisted;
       // One-off parameters.
       private long mMinLatencyMillis;
       private long mMaxExecutionDelayMillis;
       // Periodic parameters.
```



```
private boolean mIsPeriodic;
private boolean mHasEarlyConstraint;
private boolean mHasLateConstraint;
private long mIntervalMillis;
// Back-off parameters.
private long mInitialBackoffMillis = DEFAULT_INITIAL_BACKOFF_MILLIS;
private int mBackoffPolicy = DEFAULT_BACKOFF_POLICY;
/** Easy way to track whether the client has tried to set a back-off policy. */
private boolean mBackoffPolicySet = false;
public Builder(int jobId, ComponentName jobService) {
    mJobService = jobService;
    mJobId = jobId;
}
public Builder setExtras(PersistableBundle extras) {
    mExtras = extras;
    return this;
}
public Builder setRequiredNetworkType(int networkType) {
    mNetworkType = networkType;
    return this;
}
public Builder setRequiresCharging(boolean requiresCharging) {
    mRequiresCharging = requiresCharging;
    return this;
}
public Builder setRequiresDeviceIdle(boolean requiresDeviceIdle) {
    mRequiresDeviceIdle = requiresDeviceIdle;
    return this;
}
public Builder setPeriodic(long intervalMillis) {
    mIsPeriodic = true;
    mIntervalMillis = intervalMillis;
    mHasEarlyConstraint = mHasLateConstraint = true;
    return this;
}
public Builder setMinimumLatency(long minLatencyMillis) {
    mMinLatencyMillis = minLatencyMillis;
    mHasEarlyConstraint = true;
    return this;
}
public Builder setOverrideDeadline(long maxExecutionDelayMillis) {
    mMaxExecutionDelayMillis = maxExecutionDelayMillis;
    mHasLateConstraint = true;
    return this;
}
public Builder setBackoffCriteria(long initialBackoffMillis, int backoffPolicy) {
    mBackoffPolicySet = true;
    mInitialBackoffMillis = initialBackoffMillis;
    mBackoffPolicy = backoffPolicy;
    return this;
}
```

```
            public Builder setPersisted(boolean isPersisted) {
                mIsPersisted = isPersisted;
                return this;
            }
            public JobInfo build() {
                // Allow jobs with no constraints - What am I, a database?
                if (!mHasEarlyConstraint && !mHasLateConstraint && !mRequiresCharging &&
                        !mRequiresDeviceIdle && mNetworkType == NETWORK_TYPE_NONE) {
                    throw new IllegalArgumentException("You're trying to build a job with no " +
                            "constraints, this is not allowed.");
                }
                mExtras = new PersistableBundle(mExtras);    // Make our own copy.
                // Check that a deadline was not set on a periodic job.
                if (mIsPeriodic && (mMaxExecutionDelayMillis != 0L)) {
                    throw new IllegalArgumentException("Can't call setOverrideDeadline() on a " +
                            "periodic job.");
                }
                if (mIsPeriodic && (mMinLatencyMillis != 0L)) {
                    throw new IllegalArgumentException("Can't call setMinimumLatency() on a " +
                            "periodic job");
                }
                if (mBackoffPolicySet && mRequiresDeviceIdle) {
                    throw new IllegalArgumentException("An idle mode job will not respect any" +
                            " back-off policy, so calling setBackoffCriteria with" +
                            " setRequiresDeviceIdle is an error.");
                }
                return new JobInfo(this);
            }
        }
}
```

第 11 章　PMEM 内存驱动架构

在 Android 系统中，物理内存驱动 PMEM 的功能是向用户空间提供连续的物理内存区域，让 GPU 或 VPU 缓冲区共享 CPU 核心，此驱动通常用于 Android 系统的 Service 堆中。本章将探讨 Android 系统中物理内存驱动 PMEM 模块的基本知识，分析其具体原理和实现源码，为读者学习本书后面的知识打下基础。

11.1　PMEM 初步

Android 系统推出 PMEM 机制的目的是为了实现共享大尺寸连续物理内存，该机制对 DSP 和 GPU 等部件非常有用。PMEM 相当于把系统内存划分出一部分单独管理，即不被 Linux mm 管理，实际上 Linux mm 根本看不到这段内存。本节将详细讲解 PMEM 系统的基本知识。

11.1.1　什么是 PMEM

在 Android 系统中，PMEM 系统的驱动源代码保存在文件 drivers/misc/pmem.c 中，PMEM 驱动依赖于 Linux 的 Misc Device 和 Platform Driver 框架，一个系统可以有多个 PMEM，默认的是最多 10 个。在 PMEM 中暴露了如下 4 组操作。

☑ platform driver 的 probe 和 remove 操作。

☑ misc device 的 fops 接口和 vm_ops 操作。

在 Android 底层应用中，当初始化 PMEM 模块时会注册一个 platform driver，在后面的 probe（探测）操作时会创建 misc 设备文件，实现内存分配工作和初始化工作。

在 Android 底层应用中，PMEM 通过如下 3 个结构体来维护分配的共享内存。

☑ pmem_info：代表一个 PMEM 设备分配的内存块。

☑ pmem_data：代表该内存块的一个子块。

☑ pmem_region：负责把每个子块分成多个区域。

在上述 3 个结构体中，其中 pmem_data 是分配的基本单位，即每次应用层要分配一块 PMEM 内存，就会有一个 pmem_data 来表示这个被分配的内存块，实际上在 open 时，并不是 open 一个 pmem_info 表示的整个 PMEM 内存块，而是创建一个 pmem_data 以备使用。一个应用可以通过 ioctl 来分配 pmem_data 中的一个区域，并可以把它 map（绘制）到进程空间中，并不一定每次都要分配和 map 整个 pmem_data 内存块。上述 3 个数据结构的关系如图 11-1 所示。

下面将详细分析 PMEM 系统驱动的实现源码和具体用法。

11.1.2　Platform 设备基础

在 Linux 系统的设备驱动模型中，总线、设备和驱动这 3 个实体比较重要，通常总线会将设备和驱动绑定。当在系统中每注册一个设备时，会寻找与之匹配的驱动。同样的道理，每当在系统注册一个驱动时会寻找与之匹配的设备，而匹配工作是由总线完成的。

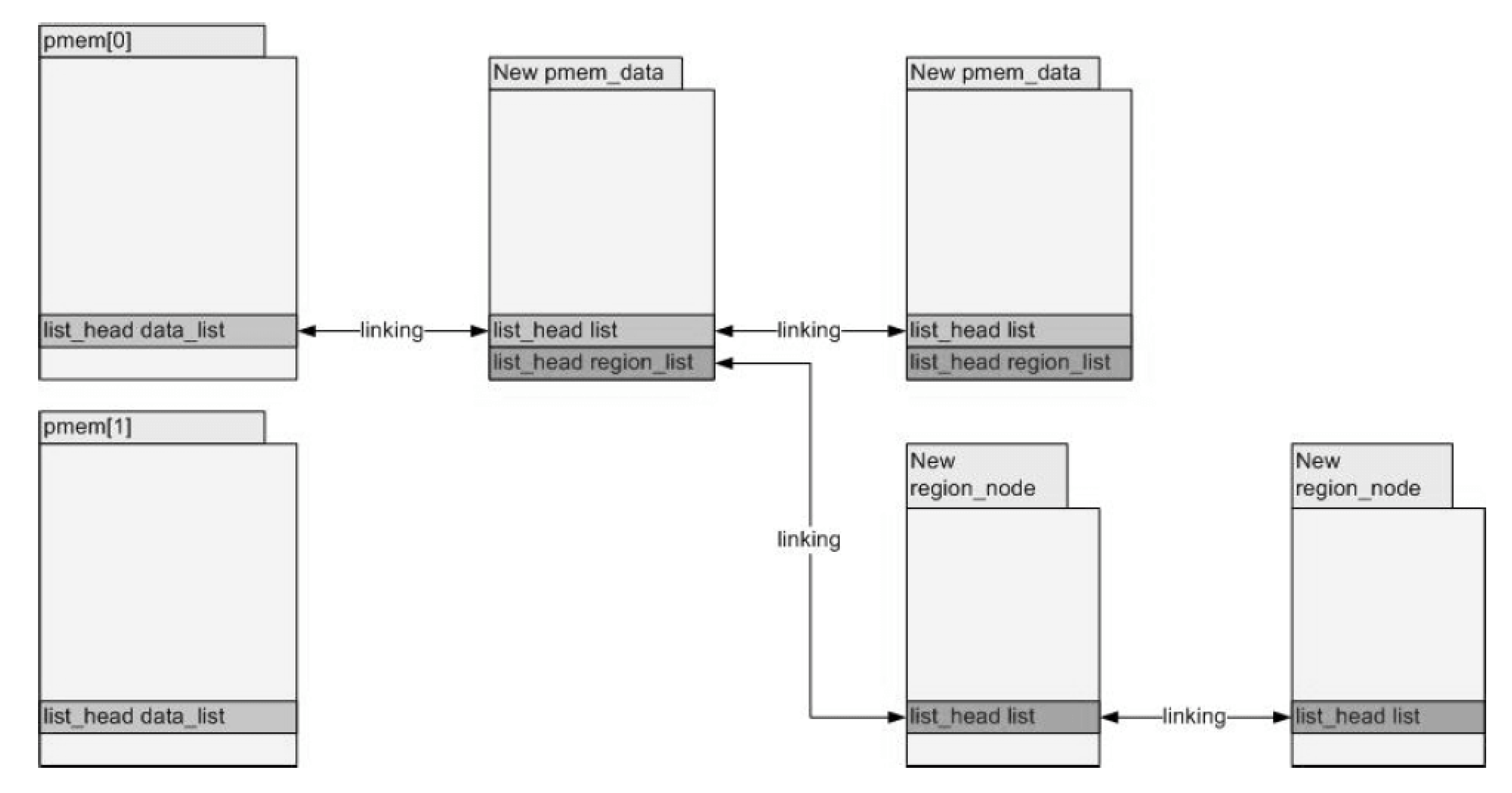


图 11-1　PMEM 结构体的关系图

在现实应用中，通常将 Linux 设备和驱动挂接在同一种总线上。当然，这对本身就依附于 PCI、USB、I2C、SPI 的设备来说自然不是问题，但是对在嵌入式系统中的设备来说，依然有如下设备不依附于此类总线。

☑　SoC 系统中集成的独立的外设控制器。

☑　挂接在 SoC 内存空间的外设。

基于上述原因，Linux 推出了一种名为“platform 总线”的虚拟总线，将与之相应的设备称为 platform_device，而将驱动称为 platform_driver。

Linux Platform Driver 机制和传统的 Device Driver 机制（通过函数 driver_register()进行注册）相比，最大的优势是 platform 机制将设备本身的资源注册进内核，这样可以由内核统一管理。当在驱动程序中使用这些资源时，可以通过 platform.device 提供的标准接口即可申请并使用。这样提高了驱动和资源管理的独立性，并且拥有较好的可移植性和安全性（这些标准接口是安全的）。

在使用 Platform 设备时，需要先了解如下几个重要的数据结构。

结构体 platform_device 用来描述设备的名称、资源信息等，该结构在文件/kernel/include/linux/platform_device.h 中定义，具体实现代码如下所示。

```
struct platform_device {
const char * name;                          //设备名
int id;                                     //设备编号
struct device dev;
u32 num_resources;                          //设备使用资源的数目
struct resource * resource;                 //设备使用资源
};
```

在结构体 platform_device 中有一个名为 struct resource * resource 的成员，struct resource 在文件 include/linux/ioport.h 中定义，具体原型如下所示。

```
struct resource {
resource_size_t start;                      //资源起始地址
resource_size_t end;                        //资源结束地址
const char *name;
unsigned long flags;                        //资源类型
struct resource *parent, *sibling, *child;
};
```

在结构体 struct resource 中，字段 start、end 和 flags 分别标明资源的开始值、结束值和类型。其中 flags 类型可以取如下值。

☑ IORESOURCE_IO
☑ IORESOURCE_MEM
☑ IORESOURCE_IRQ
☑ IORESOURCE_DMA

在结构体 struct resource 的字段中，start、end 的含义会随着 flags 的变化而变更，具体说明如下。

☑ 当 flags 为 IORESOURCE_MEM 时，start、end 分别表示该 platform_device 占据的内存的开始地址和结束地址。
☑ 当 flags 为 IORESOURCE_IRQ 时，start、end 分别表示该 platform_device 使用的中断号的开始值和结束值，如果只使用了一个中断号，开始和结束值相同。

在现实中可以有多份同种类型的资源，例如某设备占据了两个内存区域，则可以定义两个 IORESOURCE_MEM 资源。

注意：PMEM与Ashmem的区别

PMEM与Ashmem都是通过mmap实现共享的，区别是PMEM的共享区域是一段连续的物理内存，而Ashmem的共享区域在虚拟空间是连续的，物理内存却不一定连续。在现实应用中，DSP和某些设备只能在连续的物理内存上工作，这样CPU与DSP之间就需要通过PMEM来实现通信。

11.2　PMEM 驱动架构

在 Android 系统中，PMEM 驱动在 drivers/misc/pmem.c 文件中实现。

而 PMEM 驱动的设备文件在不同的设备中有不同的实现，例如可在/arch/arm/mach-msm/board-msm7x27.c 文件中实现。

本节将详细分析 PMEM 系统驱动程序的具体实现过程。

11.2.1　设备实现

对于 Android 系统的 PMEM 驱动来说，其 Device（设备）部分是通过 Platform Bus 注册实现的，具体定义代码如下所示。

```
struct platform_device mxc_android_pmem_device = {
	.name = "android_pmem",
	.id = 0,
};
```

在上述代码中，data 部分的实现代码如下所示。

```
static struct android_pmem_platform_data android_pmem_pdata = {
	.name = "pmem_adsp",
	.start = 0,
	.size = SZ_32M,
	.no_allocator = 0,
	.cached = PMEM_NONCACHE_NORMAL,
};
```

由此可见，android_pmem_pdata.start 在 fixup_mxc_board 中进行了重新计算工作，具体过程如下所示。

```
size = t->u.mem.size;
android_pmem_pdata.start =
PHYS_OFFSET + size - android_pmem_pdata.size;
```

在 Android 系统中，PMEM 模块和 ION 中的 carved-outmemory 类似，也是先预留一块内存，然后在需要时从上面分配一块内存即可。就目前情况而言，在 Android 平台上定义了如下 3 个 PMEM 模块。

☑ pmem_adsp
☑ pmem_audio
☑ pmem(mdp_pmem)

也就是说，在初始化驱动程序之前，需要先实现相应的 platform_device 部分，具体实现代码如下所示。

```
static struct android_pmem_platform_data android_pmem_adsp_pdata = {
	.name = "pmem_adsp",                              //给 adsp 使用
	.allocator_type = PMEM_ALLOCATORTYPE_BITMAP,   //都是使用 bitmap 算法，后面会讲到
	.cached = 1,
	.memory_type = MEMTYPE_EBI1,                     //内存类型都是 EBI1
};

static struct platform_device android_pmem_adsp_device = {
	.name = "android_pmem",
	.id = 1,
	.dev = { .platform_data = &android_pmem_adsp_pdata },
};

static unsigned pmem_mdp_size = MSM_PMEM_MDP_SIZE;
static int __init pmem_mdp_size_setup(char *p)
{
	pmem_mdp_size = memparse(p, NULL);
	return 0;
}
/*可以通过传参来设置 pmem mdp 的 size，其他 pmem 模块也如此*/
early_param("pmem_mdp_size", pmem_mdp_size_setup);

static unsigned pmem_adsp_size = MSM_PMEM_ADSP_SIZE;
static int __init pmem_adsp_size_setup(char *p)
{
	pmem_adsp_size = memparse(p, NULL);
	return 0;
}

early_param("pmem_adsp_size", pmem_adsp_size_setup);

static struct android_pmem_platform_data android_pmem_audio_pdata = {
	.name = "pmem_audio",          //给 audio 使用
	.allocator_type = PMEM_ALLOCATORTYPE_BITMAP,
	.cached = 0,
	.memory_type = MEMTYPE_EBI1,
};

static struct platform_device android_pmem_audio_device = {
```

```
        .name = "android_pmem",
        .id = 2,
        .dev = { .platform_data = &android_pmem_audio_pdata },
};

static struct android_pmem_platform_data android_pmem_pdata = {
        .name = "pmem",      //给 mdp 使用，Quaclomm 为什么不写成 pmem_mdp?
        .allocator_type = PMEM_ALLOCATORTYPE_BITMAP,
        .cached = 1,
        .memory_type = MEMTYPE_EBI1,
};
static struct platform_device android_pmem_device = {
        .name = "android_pmem",
        .id = 0,
        .dev = { .platform_data = &android_pmem_pdata },
};
```

这样即拥有了相应的 platform_device 实现，接下来需要找到其对应的 platform_driver 作为设备匹配，对应的实现文件是 drivers/misc/pmem.c。

11.2.2　PMEM 驱动的具体实现

下面将详细介绍 PMEM 驱动文件 drivers/misc/pmem.c 的实现过程。

（1）初始化操作。

首先定义设备结构体 pmem_driver，具体代码如下所示。

```
static struct platform_driver pmem_driver = {
        .probe = pmem_probe,
        .remove = pmem_remove,
        .driver = { .name = "android_pmem" }
};
```

然后看初始化和释放函数，具体代码如下所示。

```
static int __init pmem_init(void)
{
    /*创建 sysfs，位于/sys/kernel/ pmem_regions，以每个 PMEM 模块的名字命名，如 pmem_audio*/
    /*目录下的信息主要供用户空间查看当前 PMEM 模块的使用状况*/
    pmem_kset = kset_create_and_add(PMEM_SYSFS_DIR_NAME,
        NULL, kernel_kobj);
    if (!pmem_kset) {
        pr_err("pmem(%s):kset_create_and_add fail\n", __func__);
        return -ENOMEM;
    }
    /*寻找 platform device，接着调用 pmem_probe */
    return platform_driver_register(&pmem_driver);
}
#ifdef CONFIG_MEMORY_HOTPLUG
hotplug_memory_notifier(pmem_memory_callback, 0);
#endif
return platform_driver_register(&pmem_driver);
}
static void __exit pmem_exit(void)
```

```
{
platform_driver_unregister(&pmem_driver);
}
module_init(pmem_init);
module_exit(pmem_exit);
```

在上述代码中用到了 Linux 系统中的注册驱动函数和注销函数，这两个函数在文件/drivers/base/platform.c 中实现，具体代码如下所示。

```
void platform_driver_unregister(struct platform_driver *drv)
{
	driver_unregister(&drv->driver);
}
EXPORT_SYMBOL_GPL(platform_driver_unregister);
int __init early_platform_driver_register(struct early_platform_driver *epdrv,
					  char *buf)
{
	char *tmp;
	int n;

	/* Simply add the driver to the end of the global list.
	 * Drivers will by default be put on the list in compiled-in order.
	 */
	if (!epdrv->list.next) {
		INIT_LIST_HEAD(&epdrv->list);
		list_add_tail(&epdrv->list, &early_platform_driver_list);
	}

	/* If the user has specified device then make sure the driver
	 * gets prioritized. The driver of the last device specified on
	 * command line will be put first on the list.
	 */
	n = strlen(epdrv->pdrv->driver.name);
	if (buf && !strncmp(buf, epdrv->pdrv->driver.name, n)) {
		list_move(&epdrv->list, &early_platform_driver_list);

		/* Allow passing parameters after device name */
		if (buf[n] == '\0' || buf[n] == ',')
			epdrv->requested_id = -1;
		else {
			epdrv->requested_id = simple_strtoul(&buf[n + 1],
							     &tmp, 10);

			if (buf[n] != '.' || (tmp == &buf[n + 1])) {
				epdrv->requested_id = EARLY_PLATFORM_ID_ERROR;
				n = 0;
			} else
				n += strcspn(&buf[n + 1], ",") + 1;
		}

		if (buf[n] == ',')
			n++;

```

```
                if (epdrv->bufsize) {
                        memcpy(epdrv->buffer, &buf[n],
                                min_t(int, epdrv->bufsize, strlen(&buf[n]) + 1));
                        epdrv->buffer[epdrv->bufsize - 1] = '\0';
                }
        }

        return 0;
}
```

（2）当 Device（设备）和 Driver（驱动）匹配后将执行函数 pmem_probe()实现空间处理，此函数的功能如下。

☑　获得设备的内存空间，包括物理地址和大小。

☑　对空间的管理模块进行初始化和分区域操作。

☑　创建新的结构 pmem_data。

☑　与 pmem[]建立链表关系。

函数 pmem_probe()的具体实现代码如下所示。

```
static int pmem_probe(struct platform_device *pdev)
{
struct android_pmem_platform_data *pdata;

if (!pdev || !pdev->dev.platform_data) {
pr_alert("Unable to probe pmem!\n");
return -1;
}
pdata = pdev->dev.platform_data;

pm_runtime_set_active(&pdev->dev);
pm_runtime_enable(&pdev->dev);

return pmem_setup(pdata, NULL, NULL);
}
```

由此可见，函数 pmem_probe()的实现非常简单，只是做了一个简单的检查。

（3）在函数 pmem_probe()中调用函数 pmem_setup()将得到的各个（这里是两个）pmem_data 注册到 pmem_info 数组中，在文件 pmem.c 中限制了一次最多注册 10 个 pmem_data。函数 pmem_setup()的具体实现代码如下所示。

```
int pmem_setup(struct android_pmem_platform_data *pdata,
long (*ioctl)(struct file *, unsigned int, unsigned long),
int (*release)(struct inode *, struct file *))
{
int i, index = 0, kapi_memtype_idx = -1, id, is_kernel_memtype = 0;
/*系统对设备总的 pmem 模块数量有限制*/
if (id_count >= PMEM_MAX_DEVICES) {
pr_alert("pmem: %s: unable to register driver(%s) - no more "
"devices available!\n", __func__, pdata->name);
goto err_no_mem;
}
/*size 为 0 表示在系统初始化时并没有预留一部分内存空间给此 PMEM 模块。如果这样肯定会申请失败的*/
if (!pdata->size) {
```

```
pr_alert("pmem: %s: unable to register pmem driver(%s) - zero "
"size passed in!\n", __func__, pdata->name);
goto err_no_mem;
}

id = id_count++;
/*PMEM 通过 ID 来寻找对应的 pmem 模块*/
pmem[id].id = id;
/*表示已经分配过了*/
if (pmem[id].allocate) {
pr_alert("pmem: %s: unable to register pmem driver - "
"duplicate registration of %s!\n",
__func__, pdata->name);
goto err_no_mem;
}
/*PMEM 支持多种不同的 allocate 算法，在下面的 switch case 语句中可看到，本平台都使用默认的 bitmap 算法，
对应的 allocate type 为 PMEM_ALLOCATORTYPE_BITMAP*/
pmem[id].allocator_type = pdata->allocator_type;

for (i = 0; i < ARRAY_SIZE(kapi_memtypes); i++) {
if (!strcmp(kapi_memtypes[i].name, pdata->name)) {
if (kapi_memtypes[i].info_id >= 0) {
pr_alert("Unable to register kernel pmem "
"driver - duplicate registration of "
"%s!\n", pdata->name);
goto err_no_mem;
}
if (pdata->cached) {
pr_alert("kernel arena memory must "
"NOT be configured as 'cached'. Check "
"and fix your board file. Failing "
"pmem driver %s registration!",
pdata->name);
goto err_no_mem;
}

is_kernel_memtype = 1;
kapi_memtypes[i].info_id = id;
kapi_memtype_idx = i;
break;
}
}

/*quantum 是 bitmap 的计算单位，最小为 PAGE_SIZE，当然也可以在结构体 android_pmem_platform_data 中
自己定义大小*/
pmem[id].quantum = pdata->quantum ?: PMEM_MIN_ALLOC;
if (pmem[id].quantum < PMEM_MIN_ALLOC ||
!is_power_of_2(pmem[id].quantum)) {
pr_alert("pmem: %s: unable to register pmem driver %s - "
"invalid quantum value (%#x)!\n",
__func__, pdata->name, pmem[id].quantum);
```

```
goto err_reset_pmem_info;
}

if (pdata->start % pmem[id].quantum) {
/* bad alignment for start! */
pr_alert("pmem: %s: Unable to register driver %s - "
"improperly aligned memory region start address "
"(%#lx) as checked against quantum value of %#x!\n",
__func__, pdata->name, pdata->start,
pmem[id].quantum);
goto err_reset_pmem_info;
}
/*预留的 PMEM 模块 size 必须要以 quantum 对齐*/
if (pdata->size % pmem[id].quantum) {
/* bad alignment for size! */
pr_alert("pmem: %s: Unable to register driver %s - "
"memory region size (%#lx) is not a multiple of "
"quantum size(%#x)!\n", __func__, pdata->name,
pdata->size, pmem[id].quantum);
goto err_reset_pmem_info;
}

pmem[id].cached = pdata->cached;                    //高速缓冲标志
pmem[id].buffered = pdata->buffered;                //写缓存标志
pmem[id].base = pdata->start;
pmem[id].size = pdata->size;
strlcpy(pmem[id].name, pdata->name, PMEM_NAME_SIZE);

if (pdata->unstable) {
pmem[id].memory_state = MEMORY_UNSTABLE_NO_MEMORY_ALLOCATED;
unstable_pmem_present = UNSTABLE_UNINITIALIZED;
}

pmem[id].num_entries = pmem[id].size / pmem[id].quantum;

memset(&pmem[id].kobj, 0, sizeof(pmem[0].kobj));
pmem[id].kobj.kset = pmem_kset;

switch (pmem[id].allocator_type) {
case PMEM_ALLOCATORTYPE_ALLORNOTHING:
pmem[id].allocate = pmem_allocator_all_or_nothing;
pmem[id].free = pmem_free_all_or_nothing;
pmem[id].free_space = pmem_free_space_all_or_nothing;
pmem[id].kapi_free_index = pmem_kapi_free_index_allornothing;
pmem[id].len = pmem_len_all_or_nothing;
pmem[id].start_addr = pmem_start_addr_all_or_nothing;
pmem[id].num_entries = 1;
pmem[id].quantum = pmem[id].size;
pmem[id].allocator.all_or_nothing.allocated = 0;

if (kobject_init_and_add(&pmem[id].kobj,
```

```
&pmem_allornothing_ktype, NULL,
"%s", pdata->name))
goto out_put_kobj;

break;

case PMEM_ALLOCATORTYPE_BUDDYBESTFIT:
pmem[id].allocator.buddy_bestfit.buddy_bitmap = kmalloc(
pmem[id].num_entries * sizeof(struct pmem_bits),
GFP_KERNEL);
if (!pmem[id].allocator.buddy_bestfit.buddy_bitmap)
goto err_reset_pmem_info;

memset(pmem[id].allocator.buddy_bestfit.buddy_bitmap, 0,
sizeof(struct pmem_bits) * pmem[id].num_entries);

for (i = sizeof(pmem[id].num_entries) * 8 - 1; i >= 0; i--)
if ((pmem[id].num_entries) & 1<<i) {
PMEM_BUDDY_ORDER(id, index) = i;
index = PMEM_BUDDY_NEXT_INDEX(id, index);
}
pmem[id].allocate = pmem_allocator_buddy_bestfit;
pmem[id].free = pmem_free_buddy_bestfit;
pmem[id].free_space = pmem_free_space_buddy_bestfit;
pmem[id].kapi_free_index = pmem_kapi_free_index_buddybestfit;
pmem[id].len = pmem_len_buddy_bestfit;
pmem[id].start_addr = pmem_start_addr_buddy_bestfit;
if (kobject_init_and_add(&pmem[id].kobj,
&pmem_buddy_bestfit_ktype, NULL,
"%s", pdata->name))
goto out_put_kobj;

break;

case PMEM_ALLOCATORTYPE_BITMAP: /* 0, default if not explicit */
pmem[id].allocator.bitmap.bitm_alloc = kmalloc(
PMEM_INITIAL_NUM_BITMAP_ALLOCATIONS *
sizeof(*pmem[id].allocator.bitmap.bitm_alloc),
GFP_KERNEL);
if (!pmem[id].allocator.bitmap.bitm_alloc) {
pr_alert("pmem: %s: Unable to register pmem "
"driver %s - can't allocate "
"bitm_alloc!\n",
__func__, pdata->name);
goto err_reset_pmem_info;
}

if (kobject_init_and_add(&pmem[id].kobj,
&pmem_bitmap_ktype, NULL,
"%s", pdata->name))
goto out_put_kobj;
```

```
for (i = 0; i < PMEM_INITIAL_NUM_BITMAP_ALLOCATIONS; i++) {
pmem[id].allocator.bitmap.bitm_alloc[i].bit = -1;
pmem[id].allocator.bitmap.bitm_alloc[i].quanta = 0;
}

pmem[id].allocator.bitmap.bitmap_allocs =
PMEM_INITIAL_NUM_BITMAP_ALLOCATIONS;

pmem[id].allocator.bitmap.bitmap =
kcalloc((pmem[id].num_entries + 31) / 32,
sizeof(unsigned int), GFP_KERNEL);
if (!pmem[id].allocator.bitmap.bitmap) {
pr_alert("pmem: %s: Unable to register pmem "
"driver - can't allocate bitmap!\n",
__func__);
goto err_cant_register_device;
}
pmem[id].allocator.bitmap.bitmap_free = pmem[id].num_entries;

pmem[id].allocate = pmem_allocator_bitmap;
pmem[id].free = pmem_free_bitmap;
pmem[id].free_space = pmem_free_space_bitmap;
pmem[id].kapi_free_index = pmem_kapi_free_index_bitmap;
pmem[id].len = pmem_len_bitmap;
pmem[id].start_addr = pmem_start_addr_bitmap;

DLOG("bitmap allocator id %d (%s), num_entries %u, raw size "
"%lu, quanta size %u\n",
id, pdata->name, pmem[id].allocator.bitmap.bitmap_free,
pmem[id].size, pmem[id].quantum);
break;

case PMEM_ALLOCATORTYPE_SYSTEM:

#ifdef CONFIG_MEMORY_HOTPLUG
goto err_no_mem;
#endif

INIT_LIST_HEAD(&pmem[id].allocator.system_mem.alist);

pmem[id].allocator.system_mem.used = 0;
pmem[id].vbase = NULL;

if (kobject_init_and_add(&pmem[id].kobj,
&pmem_system_ktype, NULL,
"%s", pdata->name))
goto out_put_kobj;

pmem[id].allocate = pmem_allocator_system;
pmem[id].free = pmem_free_system;
```

```
pmem[id].free_space = pmem_free_space_system;
pmem[id].kapi_free_index = pmem_kapi_free_index_system;
pmem[id].len = pmem_len_system;
pmem[id].start_addr = pmem_start_addr_system;
pmem[id].num_entries = 0;
pmem[id].quantum = PAGE_SIZE;

DLOG("system allocator id %d (%s), raw size %lu\n",
id, pdata->name, pmem[id].size);
break;

default:
pr_alert("Invalid allocator type (%d) for pmem driver\n",
pdata->allocator_type);
goto err_reset_pmem_info;
}

pmem[id].ioctl = ioctl;
pmem[id].release = release;
mutex_init(&pmem[id].arena_mutex);
mutex_init(&pmem[id].data_list_mutex);
INIT_LIST_HEAD(&pmem[id].data_list);

pmem[id].dev.name = pdata->name;
if (!is_kernel_memtype) {
pmem[id].dev.minor = id;
pmem[id].dev.fops = &pmem_fops;
pr_info("pmem: Initializing %s (user-space) as %s\n",
pdata->name, pdata->cached ? "cached" : "non-cached");

if (misc_register(&pmem[id].dev)) {
pr_alert("Unable to register pmem driver!\n");
goto err_cant_register_device;
}
} else { /* kernel region, no user accessible device */
pmem[id].dev.minor = -1;
pr_info("pmem: Initializing %s (in-kernel)\n", pdata->name);
}

/* do not set up unstable pmem now, wait until first memory hotplug */
if (pmem[id].memory_state == MEMORY_UNSTABLE_NO_MEMORY_ALLOCATED)
return 0;

if ((!is_kernel_memtype) &&
(pmem[id].allocator_type != PMEM_ALLOCATORTYPE_SYSTEM)) {
ioremap_pmem(id);
if (pmem[id].vbase == 0) {
pr_err("pmem: ioremap failed for device %s\n",
pmem[id].name);
goto error_cant_remap;
}
```

```
}

pmem[id].garbage_pfn = page_to_pfn(alloc_page(GFP_KERNEL));

return 0;

error_cant_remap:
if (!is_kernel_memtype)
misc_deregister(&pmem[id].dev);
err_cant_register_device:
out_put_kobj:
kobject_put(&pmem[id].kobj);
if (pmem[id].allocator_type == PMEM_ALLOCATORTYPE_BUDDYBESTFIT)
kfree(pmem[id].allocator.buddy_bestfit.buddy_bitmap);
else if (pmem[id].allocator_type == PMEM_ALLOCATORTYPE_BITMAP) {
kfree(pmem[id].allocator.bitmap.bitmap);
kfree(pmem[id].allocator.bitmap.bitm_alloc);
}
err_reset_pmem_info:
pmem[id].allocate = 0;
pmem[id].dev.minor = -1;
if (kapi_memtype_idx >= 0)
kapi_memtypes[i].info_id = -1;
err_no_mem:
return -1;
}
```

（4）再看结构体文件操作类型结构体 pmem_fops，具体实现代码如下所示。

```
struct file_operations pmem_fops = {
   .release = pmem_release,
   .mmap = pmem_mmap,
   .open = pmem_open,
   .unlocked_ioctl = pmem_ioctl,
};
```

在上述结构体的实现代码中定义了 4 个函数，通过这 4 个函数可以实现对 PMEM 的使用。首先看函数 pmem_open()，此函数几乎不做任何特殊的事情，只是分配了一个 struct 结构体 pmem_data，然后将初始化之后的信息保存到 file 类型的私有数据中。函数 pmem_open()的具体实现代码如下所示。

```
static int pmem_open(struct inode *inode, struct file *file)
{
	struct pmem_data *data;
	int id = get_id(file);
	int ret = 0;
#if PMEM_DEBUG_MSGS
	char currtask_name[FIELD_SIZEOF(struct task_struct, comm) + 1];
#endif

	DLOG("pid %u(%s) file %p(%ld) dev %s(id: %d)\n",
		current->pid, get_task_comm(currtask_name, current),
		file, file_count(file), get_name(file), id);
	/*分配 struct pmem_data*/
	data = kmalloc(sizeof(struct pmem_data), GFP_KERNEL);
```

```
        if (!data) {
                printk(KERN_ALERT "pmem: %s: unable to allocate memory for "
                                "pmem metadata.", __func__);
                return -1;
        }
        data->flags = 0;
        data->index = -1;
        data->task = NULL;
        data->vma = NULL;
        data->pid = 0;
        data->master_file = NULL;
#if PMEM_DEBUG
        data->ref = 0;
#endif
        INIT_LIST_HEAD(&data->region_list);
        init_rwsem(&data->sem);
        file->private_data = data;
        INIT_LIST_HEAD(&data->list);
        mutex_lock(&pmem[id].data_list_mutex);
        list_add(&data->list, &pmem[id].data_list);
        mutex_unlock(&pmem[id].data_list_mutex);
        return ret;
}
```

当使用函数 pmem_open()处理之后，如果用户进程想要使用 PMEM，则必须通过 mmap 实现，此功能对应的是 Kernel 中的函数 pmem_mmap()。函数 pmem_mmap()的具体实现代码如下所示。

```
static int pmem_mmap(struct file *file, struct vm_area_struct *vma)
{
/*取出 open 时创建的 struct pem_data*/
struct pmem_data *data = file->private_data;
int index;
/*要映射的 size 大小*/
unsigned long vma_size = vma->vm_end - vma->vm_start;
int ret = 0, id = get_id(file);

if (!data) {
pr_err("pmem: Invalid file descriptor, no private data\n");
return -EINVAL;
}
#if PMEM_DEBUG_MSGS
char currtask_name[FIELD_SIZEOF(struct task_struct, comm) + 1];
#endif
DLOG("pid %u(%s) mmap vma_size %lu on dev %s(id: %d)\n", current->pid,
get_task_comm(currtask_name, current), vma_size,
get_name(file), id);
if (vma->vm_pgoff || !PMEM_IS_PAGE_ALIGNED(vma_size)) {
#if PMEM_DEBUG
pr_err("pmem: mmaps must be at offset zero, aligned"
" and a multiple of pages_size.\n");
#endif
return -EINVAL;
}
```

```
down_write(&data->sem);
/*如果类型为 submap，也不用再 mmap。这部分和进程间共享 PMEM 有关，也就是说当主进程做了 mmap 之后，另外一个要共享的进程就无须再 mmap 了*/
if ((data->flags & PMEM_FLAGS_SUBMAP) ||
(data->flags & PMEM_FLAGS_UNSUBMAP)) {
#if PMEM_DEBUG
pr_err("pmem: you can only mmap a pmem file once, "
"this file is already mmaped. %x\n", data->flags);
#endif
ret = -EINVAL;
goto error;
}
/*index 表示当前分配的位于 bitmap 中的索引，如果为-1 就表示未分配*/
if (data->index == -1) {
mutex_lock(&pmem[id].arena_mutex);
/*根据 ID 号从 PMEM 模块上分配一部分内存，返回在 bitmap 的索引*/
index = pmem[id].allocate(id,
vma->vm_end - vma->vm_start,
SZ_4K);
mutex_unlock(&pmem[id].arena_mutex);
/* either no space was available or an error occured */
if (index == -1) {
pr_err("pmem: mmap unable to allocate memory"
"on %s\n", get_name(file));
ret = -ENOMEM;
goto error;
}
/* store the index of a successful allocation */
data->index = index;
}
/*分配的 size 不能超过整个 PMEM 模块长度*/
if (pmem[id].len(id, data) < vma_size) {
#if PMEM_DEBUG
pr_err("pmem: mmap size [%lu] does not match"
" size of backing region [%lu].\n", vma_size,
pmem[id].len(id, data));
#endif
ret = -EINVAL;
goto error;
}
/*调用的是 pmem_start_addr_bitmap 函数，返回当前在整个 PMEM 模块中的偏移*/
vma->vm_pgoff = pmem[id].start_addr(id, data) >> PAGE_SHIFT;
/*cache 的禁止操作*/
vma->vm_page_prot = phys_mem_access_prot(file, vma->vm_page_prot);
/* PMEM_FLAGS_CONNECTED 在 ioctl 接口中会被定义，表示要共享 PMEM 内存。可以先看如果要共享内存，mmap 做了什么*/
if (data->flags & PMEM_FLAGS_CONNECTED) {
struct pmem_region_node *region_node;
struct list_head *elt;
/*插入一个 pfn 页框到用 vma 中*/
```

```
if (pmem_map_garbage(id, vma, data, 0, vma_size)) {
pr_alert("pmem: mmap failed in kernel!\n");
ret = -EAGAIN;
goto error;
}
/*根据当前有多少 region_list 做一一映射*/
list_for_each(elt, &data->region_list) {
region_node = list_entry(elt, struct pmem_region_node,
list);
DLOG("remapping file: %p %lx %lx\n", file,
region_node->region.offset,
region_node->region.len);
if (pmem_remap_pfn_range(id, vma, data,
region_node->region.offset,
region_node->region.len)) {
ret = -EAGAIN;
goto error;
}
}
/*标记当前是 submap*/
data->flags |= PMEM_FLAGS_SUBMAP;
get_task_struct(current->group_leader);
data->task = current->group_leader;
data->vma = vma;
#if PMEM_DEBUG
data->pid = current->pid;
#endif
DLOG("submmapped file %p vma %p pid %u\n", file, vma,
current->pid);
} else {
/mastermap 走如下流程。映射 vma_size 大小到用户空间*/
if (pmem_map_pfn_range(id, vma, data, 0, vma_size)) {
pr_err("pmem: mmap failed in kernel!\n");
ret = -EAGAIN;
goto error;
}
data->flags |= PMEM_FLAGS_MASTERMAP;
data->pid = current->pid;
}
vma->vm_ops = &vm_ops;
error:
up_write(&data->sem);
return ret;
}
```

由此可见，函数 pmem_mmap()的具体功能如下。

☑ 根据 mmap 大小的需求重新调整空间的管理模块，并从 Device 中获得需要的空间大小。

☑ 为获得的区域重新建立页表。

☑ 如果当前状态是 CONNECTED，需要为每一个子区域重新建立页表。

再看函数 pmem_release()，功能是在 pmem.data_list 链表中找到指向 pmem_data 结构的指针，实现链表和结构体删除操作之后及时释放内存资源。函数 pmem_release()的具体实现代码如下所示。

```
static int pmem_release(struct inode *inode, struct file *file)
{
struct pmem_data *data = file->private_data;
struct pmem_region_node *region_node;
struct list_head *elt, *elt2;
int id = get_id(file), ret = 0;                    //获取信号量

#if PMEM_DEBUG_MSGS
char currtask_name[FIELD_SIZEOF(struct task_struct, comm) + 1];
#endif
DLOG("releasing memory pid %u(%s) file %p(%ld) dev %s(id: %d)\n",
current->pid, get_task_comm(currtask_name, current),
file, file_count(file), get_name(file), id);
mutex_lock(&pmem[id].data_list_mutex);
/* if this file is a master, revoke all the memory in the connected
* files */
if (PMEM_FLAGS_MASTERMAP & data->flags) {
list_for_each(elt, &pmem[id].data_list) {
struct pmem_data *sub_data =
list_entry(elt, struct pmem_data, list);       //在 pmem.data_list 链表中找到指向 pmem_data 结构的指针
int is_master;

down_read(&sub_data->sem);
is_master = (PMEM_IS_SUBMAP(sub_data) &&
file == sub_data->master_file);
up_read(&sub_data->sem);

if (is_master)
pmem_revoke(file, sub_data);
}
}
list_del(&data->list);                                          //从双向链表中删除 data->list
mutex_unlock(&pmem[id].data_list_mutex);

down_write(&data->sem);

/* if it is not a connected file and it has an allocation, free it */
if (!(PMEM_FLAGS_CONNECTED & data->flags) && has_allocation(file)) {
mutex_lock(&pmem[id].arena_mutex);
ret = pmem[id].free(id, data->index);
mutex_unlock(&pmem[id].arena_mutex);
}

/* if this file is a submap (mapped, connected file), downref the
* task struct */
if (PMEM_FLAGS_SUBMAP & data->flags)
if (data->task) {
put_task_struct(data->task); ;                                  //释放 task struct
data->task = NULL;
}
```

```
file->private_data = NULL;

list_for_each_safe(elt, elt2, &data->region_list) {                    //删除相关的 region 结构体
region_node = list_entry(elt, struct pmem_region_node, list);
list_del(elt);
kfree(region_node);
}
BUG_ON(!list_empty(&data->region_list));

up_write(&data->sem);
kfree(data);
if (pmem[id].release)
ret = pmem[id].release(inode, file);

return ret;
}
```

在上面的代码中用到了结构体 pmem_data，在里面保存了内存块中的一个子块，具体定义代码如下所示。

```
struct pmem_data {
/*分配模式：在 no_alloc 模式位图索引设置分配的大小  */
int index;
/*定义 flags 描述  */
unsigned int flags;
/*用于保护这一数据字段，如果 mm_mmap SEM 和 SEM 一样，则先处理 MM SEM */
struct rw_semaphore sem;
/*  定义 mmaping 进程的信息  */
struct vm_area_struct *vma;
/*映射过程中的任务结构  */
struct task_struct *task;
/*  映射进程的 ID */
pid_t pid;
/*  主文件描述符  */
int master_fd;
/*主文件结构*/
struct file *master_file;
/*列出当前可用的区域，如果这是一次分配*/
struct list_head region_list;
/*定义一个链表的数据，可以访问它们的调试*/
struct list_head list;
#if PMEM_DEBUG
int ref;
#endif
};
```

（5）通过前面的操作步骤已经完成了 mmap 的执行工作，接下来用户空间就可以直接操作 pmem 了。开始分析函数 pmem_allocate_from_id()，功能是得到 PMEM 模块对应的内核虚拟地址，具体实现代码如下所示。

```
static int pmem_allocator_bitmap(const int id,
const unsigned long len,
const unsigned int align)
{
/* caller should hold the lock on arena_mutex! */
```

```
int bitnum, i;
unsigned int quanta_needed;

DLOG("bitmap id %d, len %ld, align %u\n", id, len, align);
if (!pmem[id].allocator.bitmap.bitm_alloc) {
#if PMEM_DEBUG
printk(KERN_ALERT "pmem: bitm_alloc not present! id: %d\n",
id);
#endif
return -1;
}
/*以 quantum 为单位计算要分配的内存大小*/
quanta_needed = (len + pmem[id].quantum - 1) / pmem[id].quantum;
DLOG("quantum size %u quanta needed %u free %u id %d\n",
pmem[id].quantum, quanta_needed,
pmem[id].allocator.bitmap.bitmap_free, id);
/*超过整个 pmem 模块的数量则失败*/
if (pmem[id].allocator.bitmap.bitmap_free < quanta_needed) {
#if PMEM_DEBUG
printk(KERN_ALERT "pmem: memory allocation failure. "
"PMEM memory region exhausted, id %d."
" Unable to comply with allocation request.\n", id);
#endif
return -1;
}
/*将要申请的 quanta 数量再次做一个转换，因为要考虑对齐等因素*/
bitnum = reserve_quanta(quanta_needed, id, align);
if (bitnum == -1)
goto leave;
/*找到第一个未被使用过的 bitmap 的位置*/
for (i = 0;
i < pmem[id].allocator.bitmap.bitmap_allocs &&
pmem[id].allocator.bitmap.bitm_alloc[i].bit != -1;
i++)
;
/*如果找到的位置已经超出当前的 bitmap_allocs 数量，则要重新分配更大的一块 bitm_alloc*/
if (i >= pmem[id].allocator.bitmap.bitmap_allocs) {
void *temp;
/*申请的数量比上次大一倍*/
int32_t new_bitmap_allocs =
pmem[id].allocator.bitmap.bitmap_allocs << 1;
int j;
/*申请数量不能大于当前 PMEM 模块实际的数量*/
if (!new_bitmap_allocs) { /* failed sanity check!! */
#if PMEM_DEBUG
pr_alert("pmem: bitmap_allocs number"
" wrapped around to zero! Something "
"is VERY wrong.\n");
#endif
return -1;
}
```

```
/*重新分配和指定*/
if (new_bitmap_allocs > pmem[id].num_entries) {
/* failed sanity check!! */
#if PMEM_DEBUG
pr_alert("pmem: required bitmap_allocs"
" number exceeds maximum entries possible"
" for current quanta\n");
#endif
return -1;
}

temp = krealloc(pmem[id].allocator.bitmap.bitm_alloc,
new_bitmap_allocs *
sizeof(*pmem[id].allocator.bitmap.bitm_alloc),
GFP_KERNEL);
if (!temp) {
#if PMEM_DEBUG
pr_alert("pmem: can't realloc bitmap_allocs,"
"id %d, current num bitmap allocs %d\n",
id, pmem[id].allocator.bitmap.bitmap_allocs);
#endif
return -1;
}
pmem[id].allocator.bitmap.bitmap_allocs = new_bitmap_allocs;
pmem[id].allocator.bitmap.bitm_alloc = temp;
/*只对重新分配的部分初始化*/
for (j = i; j < new_bitmap_allocs; j++) {
pmem[id].allocator.bitmap.bitm_alloc[j].bit = -1;
pmem[id].allocator.bitmap.bitm_alloc[i].quanta = 0;
}

DLOG("increased # of allocated regions to %d for id %d\n",
pmem[id].allocator.bitmap.bitmap_allocs, id);
}

DLOG("bitnum %d, bitm_alloc index %d\n", bitnum, i);

pmem[id].allocator.bitmap.bitmap_free -= quanta_needed;
pmem[id].allocator.bitmap.bitm_alloc[i].bit = bitnum;
pmem[id].allocator.bitmap.bitm_alloc[i].quanta = quanta_needed;
leave:
return bitnum;
}
```

（6）开始分析 PMEM 驱动中的 ioctl 实现，ioctl 是设备驱动程序中对设备的 I/O 通道进行管理的函数。所谓对 I/O 通道进行管理，就是对设备的一些特性进行控制，例如串口的传输波特率、马达的转速等。在 Android 平台的 PMEM 驱动程序中，提供了若干个 ioctl 的 cmd 供用户空间操作，例如获取当前申请的 len，获取 PMEM 模块的总 size 和 pmem 申请等。

首先分析函数 pmem_ioctl()，功能是处理不同的 ioctl 命令。在 Linux 系统中，ioctl 支持如下命令：

☑ PMEM_GET_PHYS：获取物理地址。

☑ PMEM_MAP：映射一段内存。
☑ PMEM_GET_SIZE：返回 pmem 分配的内存大小。
☑ PMEM_UNMAPunmap：一段内存。
☑ PMEM_ALLOCATE：分配 pmem 空间，len 是参数，如果已分配则失败。
☑ PMEM_CONNECT：将一个 pmem file 与其他相连接。
☑ PMEM_GET_TOTAL_SIZE：返回 pmem device 内存的大小。

函数 pmem_ioctl()的具体实现代码如下所示。

```
static long pmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
struct pmem_data *data = file->private_data;
int id = get_id(file);
#if PMEM_DEBUG_MSGS
char currtask_name[
FIELD_SIZEOF(struct task_struct, comm) + 1];
#endif

DLOG("pid %u(%s) file %p(%ld) cmd %#x, dev %s(id: %d)\n",
current->pid, get_task_comm(currtask_name, current),
file, file_count(file), cmd, get_name(file), id);

switch (cmd) {
case PMEM_GET_PHYS:                              //得到物理参数，如果是物理地址则表示数据长度
{
struct pmem_region region;

DLOG("get_phys\n");
down_read(&data->sem);
if (!has_allocation(file)) {
region.offset = 0;
region.len = 0;
} else {
region.offset = pmem[id].start_addr(id, data);
region.len = pmem[id].len(id, data);
}
up_read(&data->sem);

if (copy_to_user((void __user *)arg, &region,
sizeof(struct pmem_region)))
return -EFAULT;

DLOG("pmem: successful request for "
"physical address of pmem region id %d, "
"offset 0x%lx, len 0x%lx\n",
id, region.offset, region.len);

break;
}
case PMEM_MAP:                                   //映射
{
struct pmem_region region;
```

```
DLOG("map\n");
if (copy_from_user(&region, (void __user *)arg,
sizeof(struct pmem_region)))
return -EFAULT;
return pmem_remap(&region, file, PMEM_MAP);
}
break;
case PMEM_UNMAP: :                                   //解映射
{
struct pmem_region region;
DLOG("unmap\n");
if (copy_from_user(&region, (void __user *)arg,
sizeof(struct pmem_region)))
return -EFAULT;
return pmem_remap(&region, file, PMEM_UNMAP);
break;
}
case PMEM_GET_SIZE:                                  //得到大小
{
struct pmem_region region;
DLOG("get_size\n");
pmem_get_size(&region, file);
if (copy_to_user((void __user *)arg, &region,
sizeof(struct pmem_region)))
return -EFAULT;
break;
}
case PMEM_GET_TOTAL_SIZE:                            //得到总的 pmem 大小
{
struct pmem_region region;
DLOG("get total size\n");
region.offset = 0;
get_id(file);
region.len = pmem[id].size;
if (copy_to_user((void __user *)arg, &region,
sizeof(struct pmem_region)))
return -EFAULT;
break;
}
case PMEM_GET_FREE_SPACE:
{
struct pmem_freespace fs;
DLOG("get freespace on %s(id: %d)\n",
get_name(file), id);

mutex_lock(&pmem[id].arena_mutex);
pmem[id].free_space(id, &fs);
mutex_unlock(&pmem[id].arena_mutex);

DLOG("%s(id: %d) total free %lu, largest %lu\n",
get_name(file), id, fs.total, fs.largest);
```

```
if (copy_to_user((void __user *)arg, &fs,
sizeof(struct pmem_freespace)))
return -EFAULT;
break;
}

case PMEM_ALLOCATE:                                    //申请一块 pmem 内存
{
int ret = 0;
DLOG("allocate, id %d\n", id);
down_write(&data->sem);
if (has_allocation(file)) {
pr_err("pmem: Existing allocation found on "
"this file descrpitor\n");
up_write(&data->sem);
return -EINVAL;
}

mutex_lock(&pmem[id].arena_mutex);
data->index = pmem[id].allocate(id,
arg,
SZ_4K);
mutex_unlock(&pmem[id].arena_mutex);
ret = data->index == -1 ? -ENOMEM :
data->index;
up_write(&data->sem);
return ret;
}
case PMEM_ALLOCATE_ALIGNED:                            //申请对齐内存
{
struct pmem_allocation alloc;
int ret = 0;

if (copy_from_user(&alloc, (void __user *)arg,
sizeof(struct pmem_allocation)))
return -EFAULT;
DLOG("allocate id align %d %u\n", id, alloc.align);
down_write(&data->sem);
if (has_allocation(file)) {
pr_err("pmem: Existing allocation found on "
"this file descrpitor\n");
up_write(&data->sem);
return -EINVAL;
}

if (alloc.align & (alloc.align - 1)) {
pr_err("pmem: Alignment is not a power of 2\n");
return -EINVAL;
}
```

```
if (alloc.align != SZ_4K &&
(pmem[id].allocator_type !=
PMEM_ALLOCATORTYPE_BITMAP)) {
pr_err("pmem: Non 4k alignment requires bitmap"
" allocator on %s\n", pmem[id].name);
return -EINVAL;
}

if (alloc.align > SZ_1M ||
alloc.align < SZ_4K) {
pr_err("pmem: Invalid Alignment (%u) "
"specified\n", alloc.align);
return -EINVAL;
}

mutex_lock(&pmem[id].arena_mutex);
data->index = pmem[id].allocate(id,
alloc.size,
alloc.align);
mutex_unlock(&pmem[id].arena_mutex);
ret = data->index == -1 ? -ENOMEM :
data->index;
up_write(&data->sem);
return ret;
}
case PMEM_CONNECT:                              //共享 pmem 内存
DLOG("connect\n");
return pmem_connect(arg, file);
case PMEM_CLEAN_INV_CACHES:
case PMEM_CLEAN_CACHES:
case PMEM_INV_CACHES:
{
struct pmem_addr pmem_addr;

if (copy_from_user(&pmem_addr, (void __user *)arg,
sizeof(struct pmem_addr)))
return -EFAULT;

return pmem_cache_maint(file, cmd, &pmem_addr);
}
case PMEM_CACHE_FLUSH:                          //通知缓存刷新内存
{
struct pmem_region region;

if (copy_from_user(&region, (void __user *)arg,
sizeof(struct pmem_region)))
return -EFAULT;

flush_pmem_file(file, region.offset, region.len);
break;
}
```

```
default:
if (pmem[id].ioctl)
return pmem[id].ioctl(file, cmd, arg);

DLOG("ioctl invalid (%#x)\n", cmd);
return -EINVAL;
}
return 0;
}
```

在上述 ioctl 指令中，PMEM_ALLOCATE 指令能够根据 allocate 大小的需求，重新调整空间的管理模块，并从 device 中获得需要的空间。在执行 PMEM_ALLOCATE 指令时调用了函数 pmem_allocate_from_id()，此函数的具体实现已经在前面讲解过。

再看 PMEM_CONNECT 命令，用于链接需要被 connected 的文件到当前文件，也就是将两个文件映射到同一块区域中。在执行 PMEM_ALLOCATE 指令时调用了函数 pmem_connect()，这是一个 cmd 操作函数，主要用于在某个其他进程和主进程之间共享 PMEM。函数 pmem_connect()的具体实现代码如下所示。

```
static int pmem_connect(unsigned long connect, struct file *file)
{
int ret = 0, put_needed;
struct file *src_file;

if (!file) {
pr_err("pmem: %s: NULL file pointer passed in, "
"bailing out!\n", __func__);
ret = -EINVAL;
goto leave;
}
/*根据主进程的 fd 获得相对应的 file*/
src_file = fget_light(connect, &put_needed);

if (!src_file) {
pr_err("pmem: %s: src file not found!\n", __func__);
ret = -EBADF;
goto leave;
}

if (src_file == file) { /* degenerative case, operator error */
pr_err("pmem: %s: src_file and passed in file are "
"the same; refusing to connect to self!\n", __func__);
ret = -EINVAL;
goto put_src_file;
}

if (unlikely(!is_pmem_file(src_file))) {
pr_err("pmem: %s: src file is not a pmem file!\n",
__func__);
ret = -EINVAL;
goto put_src_file;
} else {
/*得到 master 的 pmem data*/
struct pmem_data *src_data = src_file->private_data;
```

```
if (!src_data) {
pr_err("pmem: %s: src file pointer has no"
"private data, bailing out!\n", __func__);
ret = -EINVAL;
goto put_src_file;
}

down_read(&src_data->sem);

if (unlikely(!has_allocation(src_file))) {
up_read(&src_data->sem);
pr_err("pmem: %s: src file has no allocation!\n",
__func__);
ret = -EINVAL;
} else {
struct pmem_data *data;
/*获得 master 分配到的内存在 bitmap 中的 index*/
int src_index = src_data->index;

up_read(&src_data->sem);

data = file->private_data;
if (!data) {
pr_err("pmem: %s: passed in file "
"pointer has no private data, bailing"
" out!\n", __func__);
ret = -EINVAL;
goto put_src_file;
}

down_write(&data->sem);
if (has_allocation(file) &&
(data->index != src_index)) {
up_write(&data->sem);

pr_err("pmem: %s: file is already "
"mapped but doesn't match this "
"src_file!\n", __func__);
ret = -EINVAL;
} else {
/*将 master 的 pmem data 数据保存到当前进程中*/
data->index = src_index;
data->flags |= PMEM_FLAGS_CONNECTED; //设置标志，会在 mmap 中用到
data->master_fd = connect;
data->master_file = src_file;

up_write(&data->sem);

DLOG("connect %p to %p\n", file, src_file);
}
}
```

```
}
put_src_file:
fput_light(src_file, put_needed);
leave:
return ret;
}
```

通过上述代码可知，此 CMD 操作的过程就是将 master 的 struct pmem_data 给了当前要共享进程的过程，其中传进去的参数为主进程打开 PMEM 的 fd，在此处被称为主进程 master。

再看 PMEM_MAP 命令，此指令是为了使用空进程要执行 remap 而设置的。在执行 PMEM_MAP 指令时调用了函数 pmem_remap()，具体功能如下。

☑　确保 map 的可行性，获得 map 请求的区域。

☑　创建新 region_node，可以将获得的区域信息保存到 region_node 中，并与 region_list 建立链表关系。

函数 pmem_remap()的具体实现代码如下所示。

```
int pmem_remap(struct pmem_region *region, struct file *file,
unsigned operation)
{
int ret;
struct pmem_region_node *region_node;
struct mm_struct *mm = NULL;
struct list_head *elt, *elt2;
int id = get_id(file);
struct pmem_data *data;

DLOG("operation %#x, region offset %ld, region len %ld\n",
operation, region->offset, region->len);

if (!is_pmem_file(file)) {
#if PMEM_DEBUG
pr_err("pmem: remap request for non-pmem file descriptor\n");
#endif
return -EINVAL;
}

/* is_pmem_file fails if !file */
data = file->private_data;

/* pmem region must be aligned on a page boundry */
if (unlikely(!PMEM_IS_PAGE_ALIGNED(region->offset) ||
!PMEM_IS_PAGE_ALIGNED(region->len))) {
#if PMEM_DEBUG
pr_err("pmem: request for unaligned pmem"
"suballocation %lx %lx\n",
region->offset, region->len);
#endif
return -EINVAL;
}

/* if userspace requests a region of len 0, there's nothing to do */
if (region->len == 0)
return 0;
```

```
/* lock the mm and data */
ret = pmem_lock_data_and_mm(file, data, &mm);
if (ret)
return 0;

/*明确指定只有 master file 才能做 remap 动作*/
if (!is_master_owner(file)) {
#if PMEM_DEBUG
pr_err("pmem: remap requested from non-master process\n");
#endif
ret = -EINVAL;
goto err;
}

/* check that the requested range is within the src allocation */
if (unlikely((region->offset > pmem[id].len(id, data)) ||
(region->len > pmem[id].len(id, data)) ||
(region->offset + region->len > pmem[id].len(id, data)))) {
#if PMEM_DEBUG
pr_err("pmem: suballoc doesn't fit in src_file!\n");
#endif
ret = -EINVAL;
goto err;
}

if (operation == PMEM_MAP) {
/*生成一个 struct pem_region_node，用来保存上层传下来的 region 信息*/
region_node = kmalloc(sizeof(struct pmem_region_node),
GFP_KERNEL);
if (!region_node) {
ret = -ENOMEM;
#if PMEM_DEBUG
pr_alert("pmem: No space to allocate remap metadata!");
#endif
goto err;
}
region_node->region = *region;
/*添加到 data 的 region_list 中*/
list_add(&region_node->list, &data->region_list);
} else if (operation == PMEM_UNMAP) {
int found = 0;
list_for_each_safe(elt, elt2, &data->region_list) {
region_node = list_entry(elt, struct pmem_region_node,
list);
if (region->len == 0 ||
(region_node->region.offset == region->offset &&
region_node->region.len == region->len)) {
list_del(elt);
kfree(region_node);
found = 1;
```

```
}
}
if (!found) {
#if PMEM_DEBUG
pr_err("pmem: Unmap region does not map any"
" mapped region!");
#endif
ret = -EINVAL;
goto err;
}
}

if (data->vma && PMEM_IS_SUBMAP(data)) {
if (operation == PMEM_MAP)
ret = pmem_remap_pfn_range(id, data->vma, data,
region->offset, region->len);
else if (operation == PMEM_UNMAP)
ret = pmem_unmap_pfn_range(id, data->vma, data,
region->offset, region->len);
}

err:
pmem_unlock_data_and_mm(data, mm);
return ret;
}
```

例如在图 11-2 中，通过 PMEM 定义了 8MB 大小的空间，而 Android 系统需要获得 1MB 的空间。

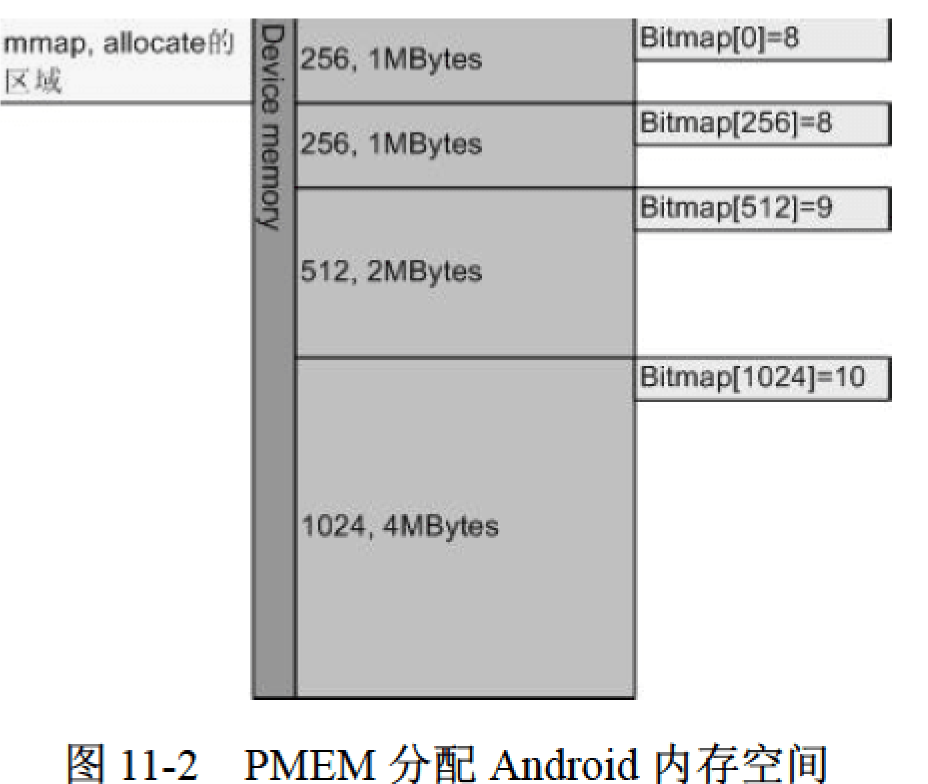


图 11-2　PMEM 分配 Android 内存空间

11.2.3　调用 PMEM 驱动的流程

（1）open 操作

具体功能是通过文件 drivers/misc/pmem.c 中的函数 pmem_open()实现的。

（2）mmap 自动调用 allocate

具体功能是通过文件 drivers/misc/pmem.c 中的如下函数实现的。

☑　pmem_allocate()。

☑　pmem_map_pfn_range()。

（3）ioctl PMEM_GET_PHYS

此处的物理地址是根据 pid 来确定的。具体功能是通过文件 drivers/misc/pmem.c 中的函数 pmem_ioctl()实现的。

（4）munmap

具体功能是通过文件 drivers/misc/pmem.c 中的函数 pmem_vma_close()实现的。

11.3　用户空间接口

在 Android 系统中，物理内存驱动 PMEM 的功能是向用户空间提供连续的物理内存区域，让 GPU 或

VPU 缓冲区共享 CPU 核心。经过本章前面内容的分析可知，PMEM 驱动会创建/dev/pme、/dev/adsp 设备，分别实现了 pmem_open、pmem_mmap、pmem_release 和 pmem_ioctl，应用层可以通过 open、mmap、close、ioctl 来操作 Pmem 设备文件。当注册为字符设备后会看到/dev/pmem_**格式的驱动，例如/dev/pmem_audio，这些驱动函数可以供用户空间操作设备。在函数 pmem_setup()中的实现代码中，展示了在用户空间中通过 open/ioctl/mmap 操作设备的如下所示的实现函数。

```
/*当前空闲的 entries*/
pmem[id].allocator.bitmap.bitmap_free = pmem[id].num_entries;
/*下面这几个函数会在用户空间通过 open/ioctl/mmap 用到*/
pmem[id].allocate = pmem_allocator_bitmap;
pmem[id].free = pmem_free_bitmap;
pmem[id].free_space = pmem_free_space_bitmap;
pmem[id].len = pmem_len_bitmap;
pmem[id].start_addr = pmem_start_addr_bitmap;
```

本节将分析上述用户控件中设备操作函数的具体实现过程。

11.3.1 释放位图内存

函数 pmem_free_bitmap()的功能是释放位图内存空间，具体实现代码如下所示。

```
static int pmem_free_bitmap(int id, int bitnum)
{
	/* caller should hold the lock on arena_mutex! */
	int i;
	char currtask_name[FIELD_SIZEOF(struct task_struct, comm) + 1];

	pr_debug("[PME][%s] pmem_free_bitmap, bitnum %d\n", pmem[id].name, bitnum);

	for (i = 0; i < pmem[id].allocator.bitmap.bitmap_allocs; i++) {
		const int curr_bit =
			pmem[id].allocator.bitmap.bitm_alloc[i].bit;

		if (curr_bit == bitnum) {
			const int curr_quanta =
				pmem[id].allocator.bitmap.bitm_alloc[i].quanta;

			bitmap_bits_clear_all(pmem[id].allocator.bitmap.bitmap,
				curr_bit, curr_bit + curr_quanta);
			pmem[id].allocator.bitmap.bitmap_free += curr_quanta;
			pmem[id].allocator.bitmap.bitm_alloc[i].bit = -1;
			pmem[id].allocator.bitmap.bitm_alloc[i].quanta = 0;
			return 0;
		}
	}
	printk(KERN_ALERT "pmem: %s: Attempt to free unallocated index %d, id"
		" %d, pid %d(%s)\n", __func__, bitnum, id,	current->pid,
		get_task_comm(currtask_name, current));

	return -1;
}
```

11.3.2　释放位图内存空间

函数 pmem_free_space_bitmap()的功能是释放指定的位图内存空间，具体实现代码如下所示。

```
static int pmem_free_space_bitmap(int id, struct pmem_freespace *fs)
{
	int i, j;
	int max_allocs;
	int alloc_start = 0;
	int next_alloc;
	unsigned long size = 0;

	//ABR
	if(id >= PMEM_MAX_DEVICES) {
		pr_err("%s: id(%d) is invalid\n", __func__, id);
		BUG_ON(true);
	}
	max_allocs = pmem[id].allocator.bitmap.bitmap_allocs;
	fs->total = 0;
	fs->largest = 0;

	for (i = 0; i < max_allocs; i++) {

		int alloc_quanta = 0;
		int alloc_idx = 0;
		next_alloc = pmem[id].num_entries;

		/*寻找最低点以开始下一步分配工作*/
		for (j = 0; j < max_allocs; j++) {
			const int curr_alloc = pmem[id].allocator.
						bitmap.bitm_alloc[j].bit;
			if (curr_alloc != -1) {
				if (alloc_start == curr_alloc)
					alloc_idx = j;
				if (alloc_start >= curr_alloc)
					continue;
				if (curr_alloc < next_alloc)
					next_alloc = curr_alloc;
			}
		}
		alloc_quanta = pmem[id].allocator.bitmap.
				bitm_alloc[alloc_idx].quanta;
		size = (next_alloc - (alloc_start + alloc_quanta)) *
				pmem[id].quantum;

		if (size > fs->largest)
			fs->largest = size;
		fs->total += size;
```

```
		if (next_alloc == pmem[id].num_entries)
			break;
		else
			alloc_start = next_alloc;
	}
	return 0;
}
```

11.3.3 获取位图占用内存

函数 pmem_len_bitmap()的功能是获取位图占用的内存，具体实现代码如下所示。

```
static unsigned long pmem_len_bitmap(int id, struct pmem_data *data)
{
	int i;
	unsigned long ret = 0;

	mutex_lock(&pmem[id].arena_mutex);

	for (i = 0; i < pmem[id].allocator.bitmap.bitmap_allocs; i++)
		if (pmem[id].allocator.bitmap.bitm_alloc[i].bit ==
				data->index) {
			ret = pmem[id].allocator.bitmap.bitm_alloc[i].quanta *
				pmem[id].quantum;
			break;
		}

	mutex_unlock(&pmem[id].arena_mutex);
#if PMEM_DEBUG
	if (i >= pmem[id].allocator.bitmap.bitmap_allocs)
		pr_alert("pmem: %s: can't find bitnum %d in "
			"alloc'd array!\n", __func__, data->index);
#endif
	return ret;
}
```

最后看函数 pmem_start_addr_bitmap()，功能是获取位图内存的开始地址，具体实现代码如下所示。

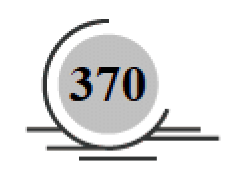
```
static unsigned long pmem_start_addr_bitmap(int id, struct pmem_data *data)
{
	return data->index * pmem[id].quantum + pmem[id].base;
}
```

11.4 实战演练——将 PMEM 加入到内核中

在 Android 系统中，PMEM 并不像 Ashmem 驱动和 Binder 驱动那样，选中之后就可以被 Android 系统所使用。PMEM 是一个 Platform 设备，只有在注册之后才可以使用。接下来将以 S3C6410 板子为例，详细介绍在底层将 PMEM 加入到内核中使用的具体流程。

（1）在内核中选中如下选项。

☑ Device Drivers --->



☑ [*] Misc devices --->

☑ [*] Android pmem allocator

（2）修改注册文件 dev.c，在里面添加如下所示的代码。

```
#ifdef CONFIG_ANDROID_PMEM
static struct android_pmem_platform_data android_pmem_pdata = {
        .name = "pmem",
        .start = PMEM_BASE,
        .size = PMEM_BASE_SIZE,
        .no_allocator = 1,
        .cached = 1,
};
static struct android_pmem_platform_data android_pmem_adsp_pdata = {
        .name = "pmem_adsp",
        .start = PMEM_ADSP_BASE,
        .size = PMEM_ADSP_BASE_SIZE,
        .no_allocator = 0,
        .cached = 0,
};
struct platform_device android_pmem_device = {
        .name = "android_pmem",
        .id = 0,
        .dev = { .platform_data = &android_pmem_pdata },
};

struct platform_device android_pmem_adsp_device = {
        .name = "android_pmem",
        .id = 1,
        .dev = { .platform_data = &android_pmem_adsp_pdata },
};
#endif
```

（3）打开驱动注册列表，在里面添加如下所示的代码。

```
static struct platform_device *smdk6410_devices[] __initdata = {
#ifdef CONFIG_ANDROID_PMEM
        &android_pmem_device,
        &android_pmem_adsp_device,
#endif
};
```

（4）分配一个物理地址，例如分配使用 128MB 中的最后 8MB，设置代码如下所示。

```
#define PMEM_BASE 0x57900000
#define PMEM_BASE_SIZE SZ_1M*4
#define PMEM_ADSP_BASE 0x57c00000
#define PMEM_ADSP_BASE_SIZE SZ_1M*4
```

（5）重新编译内核文件，具体编译方法和本书前面介绍的方法一样。

（6）修改 bootargs，目的是减少 Linux 可以管理的 MEM，例如修改为 120MB。

```
MEM=120MB
```

（7）重新启动系统，启动信息如下所示。

```
pmem: 1 init
pmem_adsp: 0 init
```

如果此时查看 dev 目录，会发现在里面新增了 pmem 目录和 pmem_adsp 目录，这样就成功地将 PMEM 加入到了内核当中。

11.5　实战演练——将 PMEM 加入到内核中

在 Android 系统中，PMEM 驱动程序是物理内存类型的驱动程序，可以用来分配物理内存。在现实应用中，PMEM 经常用在 Camera 和 Video 系统中。下面详细讲解在应用中将 PMEM 加入到内核中的具体流程。

（1）在使用 PMEM 之前，首先需要包含如下所示的头文件。

```
#include <sys/ioctl.h>
#include <binder/MemoryHeapBase.h>
#include <binder/MemoryHeapPmem.h>
#include <linux/android_pmem.h>
```

然后定义如下所示的数据结构。

```
#define PMEM_DEV "/dev/pmem0"                    //PMEM 设备的路径
#define kBufferCount 3                           //申请的 buffer 数目
sp<MemoryBase> mBuffers[kBufferCount];           //存储 PMEM buffer 的数组
int mBuffersPhys[kBufferCount];                  //存储 PMEM buffer 的物理地址
int8 *mBuffersVirt[kBufferCount];                //存储 PMEM buffer 的逻辑地址
sp<MemoryHeapBase> masterHeap;
sp<MemoryHeapPmem> mPreviewHeap;
```

（2）编写 3 个大小为 mPreviewFrameSize 的 buffer，然后获取每一个 buffer 的物理地址和逻辑地址，并同时将这 3 个 buffer 放入数组 mBuffers 中，具体实现代码如下所示。

```
Int mem_size = kBufferCount * mPreviewFrameSize;
masterHeap = new
MemoryHeapBase(PMEM_DEV,mem_size,MemoryHeapBase::NO_CACHING);
mPreviewHeap = new MemoryHeapPmem(masterHeap,MemoryHeapBase::NO_CACHING);
if (mPreviewHeap->getHeapID() >= 0) {
mPreviewHeap->slap();
masterHeap.clear();
struct pmem_region region;
int fd_pmem = 0;
fd_pmem = mPreviewHeap->getHeapID();
::ioctl(fd_pmem,PMEM_GET_PHYS,®ion);          //获取物理地址
for(int i = 0; i < kBufferCount; i++){
mBuffersPhys[i] = region.offset + i * mPreviewFrameSize;
mBuffersVirt[i] = (int8 *)mPreviewHeap->getBase() + i * mPreviewFrameSize;
mBuffers[i] = new MemoryBase(mPreviewHeap, i * mPreviewFrameSize, mPreviewFrameSize);
ssize_t offset;
size_t size;
mBuffers[i]->getMemory(&offset, &size);
LOGD("Preview buffer %d: offset: %d, size: %d.", i, offset, size);
}
}
else LOGE("Camera preview heap error: could not create master heap!");
```

对上述代码段的具体说明如下。

☑　mPreviewFrameSize：表示一帧的大小，即 byte 数。

☑　MemoryHeapBase::NO_CACHING：表示该区域不会被 cache。

☑　ioctl(fd_pmem,PMEM_GET_PHYS,®ion)：用于获取被分配的区域对应的物理地址。

☑　mBuffersPhys = region.offset + i * mPreviewFrameSize：用于获取每个 buffer 对应的物理地址。

- ☑ mBuffersVirt = (int8 *)mPreviewHeap->getBase() + i * mPreviewFrameSize：用于获取每个 buffer 对应的逻辑地址。
- ☑ mBuffers = new MemoryBase(mPreviewHeap, i * mPreviewFrameSize, mPreviewFrameSize)：表示被分配区域对应的每个 buffer 的信息。
- ☑ mBuffers->getMemory(&offset, &size)：用于获取每个 buffer 的 offset 和大小。

11.6 实战演练——PMEM 在 Camera 中的应用

在 Android 系统中，文件 hardware/mx5x/libcamera/Camera_pmem.cpp 演示了使用 PMEM 的过程。其中构造函数 memAllocator()的具体实现代码如下所示。

```
//构造函数，调用 pmem 分配时将传入两个值，分别是 bufCount 和 bufSize
PmemAllocator:memAllocator(int bufCount, int bufSize):
    err_ret(0), mFD(0),mTotalSize(0),mBufCount(bufCount),mBufSize(bufSize),     //初始化变量
    mVirBase(NULL),mPhyBase(NULL)
{
    LOG_FUNCTION_NAME;
//将所有槽的标记清 0，将 pmem 默认分为 MAX_SLOT 份，这个分法个人认为不是很严谨，容易溢出
    memset(mSlotAllocated, 0, sizeof(bool)*MAX_SLOT);

    int err;
    struct pmem_region region;
    mFD = open(PMEM_DEV, O_RDWR);              //打开 pmem 设备，就是上面驱动中注册的 misc 设备
    if (mFD < 0) {
        LOGE("Error!PmemAllocator constructor");
        err_ret = -1;
        return;
    }

    err = ioctl(mFD, PMEM_GET_TOTAL_SIZE, &region);                //得到总的 pmem 大小
    if (err == 0)
    {
        LOGE("Info!get pmem total size %d",(int)region.len);
    }
    else
    {
       LOGE("Error!Cannot get total length in PmemAllocator constructor");
       err_ret = -1;
       return;
    }

    mBufSize = (bufSize + DEFAULT_PMEM_ALIGN-1) & ~(DEFAULT_PMEM_ALIGN-1);//内存块对齐

    mTotalSize = mBufSize*bufCount;                       //要申请的大小
    if((mTotalSize > region.len)||(mBufCount > MAX_SLOT)) { //判断 pmem 是否能满足需求大小
       LOGE("Error!Out of PmemAllocator capability");
    }
    else
    {
```

```
        //映射申请的大小，由上面 pmem 分析得知，它将会把用户空间映射到 pmem 区域
        uint8_t *virtualbase = (uint8_t*)mmap(0, mTotalSize,
            PROT_READ|PROT_WRITE, MAP_SHARED, mFD, 0);

        if (virtualbase == MAP_FAILED) {
            LOGE("Error!mmap(fd=%d, size=%u) failed (%s)",
                    mFD, (unsigned int)mTotalSize, strerror(errno));
            return;
        }

        memset(&region, 0, sizeof(region));

        if (ioctl(mFD, PMEM_GET_PHYS, &region) == -1)     //得到映射的物理参数，如物理地址、映射长度
        {
           LOGE("Error!Failed to get physical address of source!\n");
           munmap(virtualbase, mTotalSize);
           return;
        }
        mVirBase = (void *)virtualbase;                    //赋值给全局变量
        mPhyBase = region.offset;                          //就是刚才得到的物理参数中的映射信息
        LOGV("Allocator total size %d, vir addr 0x%x, phy addr 0x%x",mTotalSize,mVirBase,mPhyBase);
    }
}
```

析构函数~PmemAllocator()的具体实现代码如下所示。

```
PmemAllocator::~PmemAllocator()                            //析构函数
{
    LOG_FUNCTION_NAME;
    for(int index=0;index < MAX_SLOT;index ++) {
        if(mSlotAllocated[index]) {
            LOGE("Error!Cannot deinit PmemAllocator before all memory back to allocator");
        }
    }
    if(mVirBase) {
        munmap(mVirBase, mTotalSize);
    }
    if(mFD) {
        close(mFD);
    }

}
```

再看函数 allocate()，功能是从构造函数中申请的 pmem 中分配一块 bufSize 大小的内存，具体实现代码如下所示。

```
int PmemAllocator::allocate(struct picbuffer *pbuf, int size)
{
    LOG_FUNCTION_NAME;
    if((!mVirBase)||(!pbuf)||(size>mBufSize)) {           //一般 size 等于 mBufSize
        LOGE("Error!No memory for allocator");
        return -1;
    }
    for(int index=0;index < MAX_SLOT;index ++) {
        if(!mSlotAllocated[index]) {                     //找到还没被使用的一块
```

```
                LOGE("Free slot %d for allocating mBufSize %d request size %d",
                                  index,mBufSize,size);
                pbuf->virt_start= (unsigned char *)mVirBase+index*mBufSize;
                pbuf->phy_offset= mPhyBase+index*mBufSize;
                pbuf->length= mBufSize;
                mSlotAllocated[index] = true;                //置上被使用的标记
                return 0;
            }
        }
        return -1;
}
```

函数 deAllocate()的功能是删除 allocate 分配的 buffer，具体实现代码如下所示。

```
int PmemAllocator::deAllocate(struct picbuffer *pbuf)
{
    LOG_FUNCTION_NAME;
    if((!mVirBase)||(!pbuf)) {
        LOGE("Error!No memory for allocator");
        return -1;
    }
    int nSlot = ((unsigned int)pbuf->virt_start- (unsigned int)mVirBase)/mBufSize;
    if((nSlot<MAX_SLOT)&&(mSlotAllocated[nSlot])) {
        LOGE("Info!deAllocate for slot %d",nSlot);
        mSlotAllocated[nSlot] = false;
        return 0;
    }
    else{
        LOGE("Error!Not a valid buffer");
        return -1;
    }
}
```

11.7　实战演练——PMEM 的移植与测试

在现实应用中，通常需要在 Video 设备中分配大块连续物理内存，此时就需要移植 Android 中的 PMEM 系统，具体实现流程如下。

（1）编写驱动文件，为了简便性，可以直接从 Android 的 Linux 内核中复制，并配置好 makefile 和 config 文件。

（2）在文件 devs.c 中添加如下所示的代码。

```
#ifdef CONFIG_ANDROID_PMEM
#include <linux/android_pmem.h>
#include <linux/memblock.h>
#endif
    #ifdef CONFIG_ANDROID_PMEM

    static struct android_pmem_platform_data pmem_pdata = {
        .name = "pmem",
        .no_allocator = 1,
        .cached = 1,
        .start = 0,
```

```
        .size = 0,
    };

    struct platform_device pmem_device = {
        .name = "android_pmem",
        .id = 0,
        .dev = { .platform_data = &pmem_pdata },
    };

    void __init s5p_pmem_reserve_mem(phys_addr_t base, unsigned int size)
    {
            if (memblock_remove(base, size)) {
                printk(KERN_ERR "Failed to reserve memory for PMEM device (%ld bytes at 0x%08lx)\n",
size, (unsigned long)base);
                base = 0;
                return;
            }
            printk(KERN_INFO "reserve memory for PMEM device (%ld bytes at 0x%08lx)\n",
                    size, (unsigned long)base);
            pmem_pdata.start = base;
            pmem_pdata.size = size;
    }
    #endif
```

（3）在文件 devs.h 中添加如下所示的代码。

```
#ifdef CONFIG_ANDROID_PMEM
extern struct platform_device pmem_device;
extern void __init s5p_pmem_reserve_mem(phys_addr_t base, unsigned int size);
#endif
```

（4）在文件 mach-smdkv210.c 中添加如下所示的代码。

```
static void __init smdkv210_reserve(void)
{
    s5p_mfc_reserve_mem(0x3b800000, 36 << 20, 0x3dc00000, 36 << 20);
#ifdef CONFIG_ANDROID_PMEM
    s5p_pmem_reserve_mem(0x3b000000, 8 << 20);
#endif
}
```

（5）在 static struct platform_device *smdkv210_devices[]中加入如下所示的变量。

```
#ifdef CONFIG_ANDROID_PMEM
&pmem_device,
#endif
```

（6）最后编写测试程序 test.c，具体实现代码如下所示。

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <sys/mman.h>
```

```
#include <assert.h>
#include <linux/videodev2.h>
#include <linux/fb.h>
#include <pthread.h>
#include <poll.h>
#include <semaphore.h>
#include "android_pmem.h"

int main()
{
	int pmem_fd;
	void *pmem_base;
	unsigned int size;
	struct pmem_region region;

	pmem_fd = open("/dev/pmem", O_RDWR, 0);//打开设备，为了操作硬件引擎，要 noncache 的
	printf("pmem_fd:%d\n", pmem_fd);
	if (pmem_fd >= 0)
	{
	    if (ioctl(pmem_fd, PMEM_GET_TOTAL_SIZE, &region) < 0) //获取全部空间
		    perror("PMEM_GET_TOTAL_SIZE failed\n");


		size = region.len;
		printf("region.len:0x%08x offset:0x%08x\n",region.len, region.offset);
		pmem_base = mmap(0, size, PROT_READ|PROT_WRITE, MAP_SHARED, pmem_fd, 0);//mmap 操作

		if (pmem_base == MAP_FAILED)
		{	pmem_base = 0;
		    close(pmem_fd);
		    pmem_fd = -1;
			perror("mmap pmem error!\n");
		}
		printf("pmem_base:0x%08x\n", pmem_base);
	}

	if ( ioctl(pmem_fd, PMEM_GET_PHYS, &region) < 0)
	{//获取物理地址
		perror("PMEM_GET_PHYS failed\n");
	}
	printf("region:0x%08x\n",region.offset);

	if(munmap(pmem_base, size) < 0)
	{
		perror("munmap error\n");
	}
	close(pmem_fd);
	return 0;
}
```

接下来就可以使用脚本文件 build.sh 来编译和安装上述驱动，在 Linux 端执行上述测试命令后会输出获取的信息。

第 12 章　调试机制驱动 Ram Console

Ram Console 是 Android 系统中的一个调试机制驱动，可以帮助程序员实现程序调试功能。Android 系统为了实现具体的调试功能，允许将基于 RAM 的 Buffer 调试日志信息写入到这个设备中。本章将详细讲解 Android 系统中调试机制驱动 Ram Console 的基本架构知识，为读者学习本书后面的知识打下基础。

12.1　Ram Console 介绍

Ram Console 驱动是一个控制台驱动的框架，它提供了一种可以辅助调试的内核机制。为了提供调试功能，Android 允许将调试日志信息写入一个被称为 Ram Console 的设备中，它是一个基于 RAM 的 Buffer。众所周知，在使用 Eclipse 开发 Android 应用程序时，在 Eclipse 界面底部的 Console 中会输出显示调试信息，如图 12-1 所示。

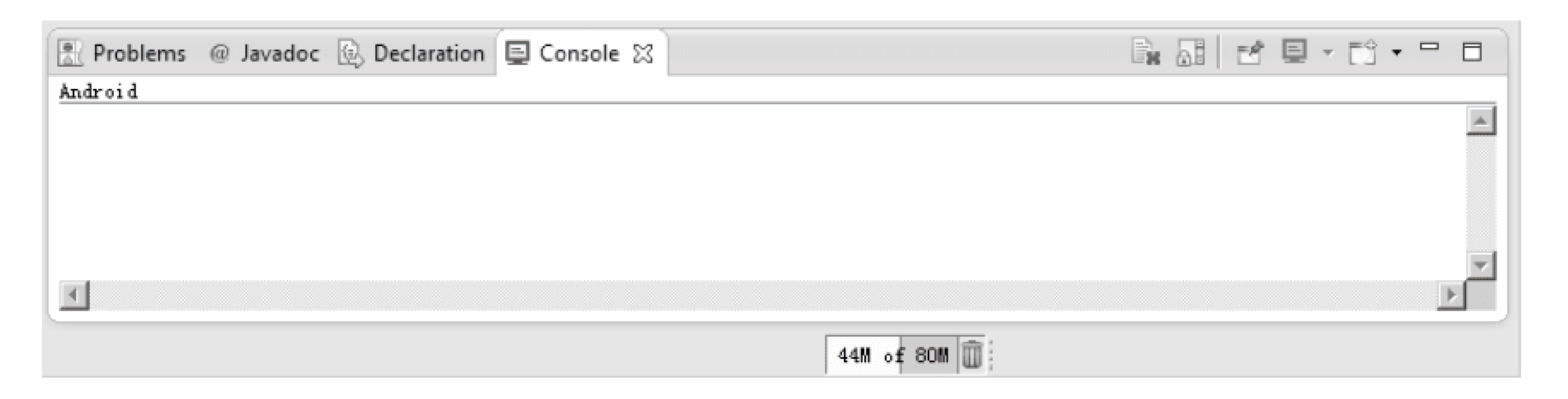


图 12-1　Eclipse 的 Console 界面

在 Android 系统中，Ram Console 驱动系统能够用一段物理内存虚拟出一个 Console 设备，这样在 printk（输出）时可以把调试信息写入 RAM 块中，最后通过/proc 文件系统输出。这样经过上述处理流程之后，会以可视化的信息将调试信息显示在开发者面前，实现了辅助调试功能。

由此可见，Ram Console 类似于普通的串口 Console，在函数 printk()内部的实现都是向已注册和打开的 Console 输出信息。Console 不但可以基于串口实现，而且也可以基于内存实现，区别是数据流的流向。在 Ram Console 实现过程中会生成 proc/last_kmsg 文件，该文件在程序调试时被用到，例如当系统发生 panic 重启情况时，该文件可以在现场保留（内存只要不掉电，其保存的信息就不会丢失）。

12.2　实现 Ram Console

在 Android 系统中，Ram Console 与用户空间之间的接口是 proc 文件系统，在 proc 中使用 last_kmsg 文件来表示 kernel 最后输出的结果信息。Ram Console 驱动的实现文件如下所示。

- ☑　drivers/staging/android/ram_console.h
- ☑　drivers/staging/android/ram_console.c

本节将详细分析上述文件的具体实现过程。

12.2.1　定义结构体 ram_console_platform_data

文件 ram_console.h 的功能是定义结构体 ram_console_platform_data，具体实现代码如下所示。

```
#ifndef _INCLUDE_LINUX_PLATFORM_DATA_RAM_CONSOLE_H_
#define _INCLUDE_LINUX_PLATFORM_DATA_RAM_CONSOLE_H_
struct ram_console_platform_data {
        const char *bootinfo;
};
#endif /* _INCLUDE_LINUX_PLATFORM_DATA_RAM_CONSOLE_H_ */
```

12.2.2　实现具体功能

文件 ram_console.c 的具体实现流程如下。

（1）首先定义结构体 ram_console_buffer，具体实现代码如下所示。

```
struct ram_console_buffer {
uint32_t sig;
uint32_t start;
uint32_t size;
uint8_t data[0];
};
```

在上述代码中，结构体 ram_console_buffer 表示一个 Ram Console 设备缓冲区，各个参数的具体说明如下。

- ☑ sig：表示在程序中主要指向 RAM_CONSOLE_SIG 宏。
- ☑ start：表示缓冲区的开始位置。
- ☑ size：表示缓冲区的大小。
- ☑ data：表示具体的数据。

（2）再看如下代码。

```
#ifdef CONFIG_ANDROID_RAM_CONSOLE_EARLY_INIT
console_initcall(ram_console_early_init);
#else
module_init(ram_console_module_init);
#endif
late_initcall(ram_console_late_init);
```

上述代码的功能是，如果是基于 RAM 的缓冲区，则调用初始化函数 ram_console_early_init()来初始化；否则，通过函数 ram_console_module_init()调用 platform_driver_register 来注册一个 Ram Console 驱动 ram_console_driver。函数 ram_console_module_init()的具体实现代码如下所示。

```
static int __init ram_console_module_init(void)
{
int err;
err = platform_driver_register(&ram_console_driver);
return err;
}
```

（3）结构体 ram_console_driver 的功能是设置设备驱动名称，通过 probe 函数 ram_console_driver_probe()基于 platformdriver 框架来初始化一个 platformdriver。结构体 ram_console_driver 的具体实现代码如下所示。

```
static struct platform_driver ram_console_driver = {
.probe = ram_console_driver_probe,
.driver = {
```

```
.name = "ram_console",
},
};
```

函数 ram_console_driver_probe()的具体实现代码如下所示。

```
static int ram_console_driver_probe(struct platform_device *pdev)
{
struct resource *res = pdev->resource;
size_t start;
size_t buffer_size;
void *buffer;
if (res == NULL || pdev->num_resources != 1 ||
!(res->flags & IORESOURCE_MEM)) {
printk(KERN_ERR "ram_console: invalid resource, %p %d flags "
"%lx\n", res, pdev->num_resources, res ? res->flags : 0);
return -ENXIO;
}
buffer_size = res->end - res->start + 1;
start = res->start;
printk(KERN_INFO "ram_console: got buffer at %zx, size %zx\n",
start, buffer_size);
//通过 ioremap 将保留的物理内存映射到内核的地址空间中（代码是不会直接访问物理内存的，必须要经过页表的
转换）
buffer = ioremap(res->start, buffer_size);
if (buffer == NULL) {
printk(KERN_ERR "ram_console: failed to map memory\n");
return -ENOMEM;
}
return ram_console_init(buffer, buffer_size, NULL/* allocate */);
}
```

（4）如果是基于 RAM 的缓冲区实现，则调用函数 ram_console_early_init()来实现初始化操作，具体实现代码如下所示。

```
static int __init ram_console_early_init(void)
{
return ram_console_init((struct ram_console_buffer *)
CONFIG_ANDROID_RAM_CONSOLE_EARLY_ADDR,
CONFIG_ANDROID_RAM_CONSOLE_EARLY_SIZE,
ram_console_old_log_init_buffer);
}
```

通过前面的实现代码可知，无论采用哪一种方式来实现初始化工作，最后都会调用函数 ram_console_init()来执行初始化操作，具体的区别是当通过函数 ram_console_early_init()进行初始化时，会构建一个 ram_console_buffer 缓冲区并传入到 ram_console_init 中进行初始化。也就是需要创建一个 ram_console_buffer 缓冲区，并对其执行初始化和赋值操作，最后通过 register_console(&ram_console)来注册 ram_console。

（5）函数 register_console()是 Linux 系统中的一个串口注册函数，功能是首先检查要注册的串口的合法性，然后将其加入 console_drivers 链表，在链表中可能存在多个合法的串口设备。在此时可以查看 flags 标志的值，如果 flags & CON_ENABLED==1 表示现在可使用的串口，显然只能有一个设备设置此标志，该标志在 register_console()函数中被设置。函数 register_console()的具体实现代码如下所示。

```
void register_console(struct console *newcon)
{
```

```
        int i;
        unsigned long flags;
        struct console *bcon = NULL;
        if (console_drivers && newcon->flags & CON_BOOT) {     //注册的是引导控制台
                for_each_console(bcon) { //遍历全局 console_drivers 数组
                        if (!(bcon->flags & CON_BOOT)) {    //判断是否已经有引导控制台，如已有就直接退出
                                printk(KERN_INFO "Too late to register bootconsole %s%d\n",newcon->name, newcon->
index);
                                return;
                        }
                }
        }

        if (console_drivers && console_drivers->flags & CON_BOOT)  //如果注册的是引导控制台
                bcon = console_drivers;    //让 bcon 指向全局 console_drivers
        if (<span style="BACKGROUND-COLOR: #ffd700">preferred_console</span> < 0 || bcon || !console_
drivers)
                <span style="BACKGROUND-COLOR: #ffd700">preferred_console</span> = selected_console;
        //设置<span style="BACKGROUND-COLOR: #ffd700">preferred_console</span>为 uboot 命令选择的
selected_console(索引)
        if (newcon->early_setup)  //存在 early_setup()函数
                newcon->early_setup();    //则调用 early_setup()函数
        if (<span style="BACKGROUND-COLOR: #ffd700">preferred_console</span> < 0) {
                if (newcon->index < 0)
                        newcon->index = 0;
                if (newcon->setup == NULL ||newcon->setup(newcon, NULL) == 0) {
                        newcon->flags |= CON_ENABLED;
                        if (newcon->device) {
                                newcon->flags |= CON_CONSDEV;
                                <span style="BACKGROUND-COLOR: #ffd700">preferred_console</span> = 0;
                        }
                }
        }       //遍历全局 console_cmdline 找到匹配的
        for (i = 0; i < MAX_CMDLINECONSOLES && console_cmdline[i].name[0];i++) {
                if (strcmp(console_cmdline[i].name, newcon->name) != 0)//比较名字
                        continue;
                if (newcon->index >= 0 &&newcon->index != console_cmdline[i].index)     //比较次设备号
                        continue;
                if (newcon->index < 0)     //若没有指定设备号
                        newcon->index = console_cmdline[i].index;
#ifdef CONFIG_A11Y_BRAILLE_CONSOLE
                if (console_cmdline[i].brl_options) {
                        newcon->flags |= CON_BRL;

        braille_register_console(newcon,console_cmdline[i].index,console_cmdline[i].options,console_cmdline[i].b
rl_options);
                        return;
                }
#endif
                if (newcon->setup &&newcon->setup(newcon, console_cmdline[i].options) != 0)       //存在 setup()方
法调用该方法
```

```
                break;
            newcon->flags |= CON_ENABLED;  //设置标志为 CON_ENABLE（这个在 printk 调用中会用到）
            newcon->index = console_cmdline[i].index;      //设置索引号
            if (i == selected_console) {                   //索引号和 uboot 指定的 console 一样
                newcon->flags |= CON_CONSDEV;              //设置标志 CON_CONSDEV（全局 console_drivers
链表中靠前）
                <span style="BACKGROUND-COLOR: #ffd700">preferred_console</span> = selected_console;
                                                           //设置 preferred
            }
            break;
        }   //for 循环作用大致是查看注册的 console 是否是 uboot 指向的引导 console，如果是则设置相关标志和
<span style="BACKGROUND-COLOR: #ffd700">preferred_console</span>
        if (!(newcon->flags & CON_ENABLED))
            return;
        if (bcon && ((newcon->flags & (CON_CONSDEV | CON_BOOT)) == CON_CONSDEV))   //防止重复打印
            newcon->flags &= ~CON_PRINTBUFFER;
        acquire_console_sem();
        if ((newcon->flags & CON_CONSDEV) || console_drivers == NULL) {      //如果是 preferred 控制台
            newcon->next = console_drivers;
            console_drivers = newcon;          //添加进全局 console_drivers 链表前面位置(printk 中会遍历该表
调用合适的 console 的 write 方法打印信息)
            if (newcon->next)
                newcon->next->flags &= ~CON_CONSDEV;
        } else {                               //如果不是 preferred 控制台
            newcon->next = console_drivers->next;
            console_drivers->next = newcon;    //添加全局 console_drivers 链表后面位置
        }
        if (newcon->flags & CON_PRINTBUFFER) {
            spin_lock_irqsave(&logbuf_lock, flags);
            con_start = log_start;
            spin_unlock_irqrestore(&logbuf_lock, flags);
        }
        release_console_sem();
        if (bcon && ((newcon->flags & (CON_CONSDEV | CON_BOOT)) == CON_CONSDEV)) {
            printk(KERN_INFO "console [%s%d] enabled, bootconsole disabled\n",newcon->name, newcon->
index);
            for_each_console(bcon)
                if (bcon->flags & CON_BOOT)
                    unregister_console(bcon);
        } else {
            printk(KERN_INFO "%sconsole [%s%d] enabled\n",(newcon->flags & CON_BOOT) ? "boot" : "" ,
newcon->name, newcon->index);
        }
}
EXPORT_SYMBOL(register_console);
```

另外，如果是基于 RAM 的缓冲区，则调用函数 console_verbose()显示详细的信息。函数 console_verbose()的具体实现代码如下所示。

```
static inline void console_verbose(void)
{
        if (console_loglevel)
```

```
		console_loglevel = 15;
}
```

函数 ram_console_init()用于执行初始化操作，具体实现代码如下所示。

```
static int __init ram_console_init(struct ram_console_buffer *buffer,
size_t buffer_size, char *old_buf)
{
#ifdef CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION
int numerr;
uint8_t *par;
#endif
ram_console_buffer = buffer;
ram_console_buffer_size =
buffer_size - sizeof(struct ram_console_buffer);
if (ram_console_buffer_size > buffer_size) {
pr_err("ram_console: buffer %p, invalid size %zu, datasize %zu\n",
buffer, buffer_size, ram_console_buffer_size);
return 0;
}
#ifdef CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION
ram_console_buffer_size -= (DIV_ROUND_UP(ram_console_buffer_size,
ECC_BLOCK_SIZE) + 1) * ECC_SIZE;
if (ram_console_buffer_size > buffer_size) {
pr_err("ram_console: buffer %p, invalid size %zu, "
"non-ecc datasize %zu\n",
buffer, buffer_size, ram_console_buffer_size);
return 0;
}
ram_console_par_buffer = buffer->data + ram_console_buffer_size;
/* first consecutive root is 0
* primitive element to generate roots = 1
*/
ram_console_rs_decoder = init_rs(ECC_SYMSIZE, ECC_POLY, 0, 1, ECC_SIZE);
if (ram_console_rs_decoder == NULL) {
printk(KERN_INFO "ram_console: init_rs failed\n");
return 0;
}
ram_console_corrected_bytes = 0;
ram_console_bad_blocks = 0;
par = ram_console_par_buffer +
DIV_ROUND_UP(ram_console_buffer_size, ECC_BLOCK_SIZE) * ECC_SIZE;
numerr = ram_console_decode_rs8(buffer, sizeof(*buffer), par);
if (numerr > 0) {
printk(KERN_INFO "ram_console: error in header, %d\n", numerr);
ram_console_corrected_bytes += numerr;
} else if (numerr < 0) {
printk(KERN_INFO
"ram_console: uncorrectable error in header\n");
ram_console_bad_blocks++;
}
#endif
if (buffer->sig == RAM_CONSOLE_SIG) {
```

```
if (buffer->size > ram_console_buffer_size
|| buffer->start > buffer->size)
printk(KERN_INFO "ram_console: found existing invalid "
"buffer, size %d, start %d\n",
buffer->size, buffer->start);
else {
printk(KERN_INFO "ram_console: found existing buffer, "
"size %d, start %d\n",
buffer->size, buffer->start);
ram_console_save_old(buffer, old_buf);
}
} else {
printk(KERN_INFO "ram_console: no valid data in buffer "
"(sig = 0x%08x)\n", buffer->sig);
}
buffer->sig = RAM_CONSOLE_SIG;
buffer->start = 0;
buffer->size = 0;
register_console(&ram_console);
#ifdef CONFIG_ANDROID_RAM_CONSOLE_ENABLE_VERBOSE
console_verbose();
#endif
return 0;
}
```

（6）在前面用到了结构体 ram_console，具体实现代码如下所示。

```
static struct console ram_console = {
.name = 'ram',
.write = ram_console_write,
.flags = CON_PRINTBUFFER |CON_ENABLED,
.index = -1;
};
```

上述代码中指定了名称、写操作函数、flags 和 index 等信息。

（7）函数 ram_console_late_init()的具体实现流程如下。

- ☑ 如果定义了 CONFIG_ANDROID_RAM_CONSOLE_EARLY_INIT，那么就分配空间给 ram_console_old_log，并复制结构体 ram_console_old_log_init_buffer 中的数据至 ram_console_old_log 中，这样做的目的是执行初始化操作。
- ☑ 通过函数 create_proc_entry()创建 last_kmsg 目录项。
- ☑ 指定 file_operations 为 ram_console_file_ops。

函数 ram_console_late_init()的具体实现代码如下所示。

```
static int __init ram_console_late_init(void)
{
struct proc_dir_entry *entry;
if (ram_console_old_log == NULL)
return 0;
#ifdef CONFIG_ANDROID_RAM_CONSOLE_EARLY_INIT
ram_console_old_log = kmalloc(ram_console_old_log_size, GFP_KERNEL);
if (ram_console_old_log == NULL) {
printk(KERN_ERR
"ram_console: failed to allocate buffer for old log\n");
```

```
ram_console_old_log_size = 0;
return 0;
}
memcpy(ram_console_old_log,
ram_console_old_log_init_buffer, ram_console_old_log_size);
#endif
entry = create_proc_entry("last_kmsg", S_IFREG | S_IRUGO, NULL);
if (!entry) {
printk(KERN_ERR "ram_console: failed to create proc entry\n");
kfree(ram_console_old_log);
ram_console_old_log = NULL;
return 0;
}
entry->proc_fops = &ram_console_file_ops;
entry->size = ram_console_old_log_size;
return 0;
}
```

在上述代码中用到了函数 create_proc_entry()，功能是创建一个一般的 proc 文件，其中 name 是文件名，例如 hello，mode 是指文件模式，parent 是指要创建的 proc 文件的父目录（如 parent = NULL 则创建在/proc 目录下）。

```
struct proc_dir_entry *create_proc_entry( const char *name,   mode_t mode,
                        struct proc_dir_entry *parent );
```

在函数 ram_console_late_init()中，指定 file_operations 为 ram_console_file_ops 的定义代码如下所示。

```
static struct file_operations ram_console_file_ops = {
  .owner = THIS_MODULE,
  .read = ram_console_read_old,
};
```

在上述代码中，指定了用操作函数 ram_console_read_old()来读取 ram_console_old_log 的信息。函数 ram_console_read_old()的具体实现代码如下所示。

```
static ssize_tram_console_read_old(struct file *file, char __user *buf, size_t len,loff_t *offset)
{
loff_t pos = *offset;
ssize_t count;
if (pos >=ram_console_old_log_size)
return 0;
count = min(len,(size_t)(ram_console_old_log_size – pos));
//复制数据到用户空间
if (copy_to_user(buf,ram_console_old_log + pos, count))
return -EFAULT;
//改变偏移量
*offset += count;
return count;
}
```

在上述实现代码中，首先判断读取的偏移量是否大于 ram_console_old_log_size，如果大于则不能读取。然后计算要读取数据的计数并存储于 count 中，并将指定的数据复制到用户空间中。最后改变偏移量的大小。

（8）接下来看读取操作函数的具体实现，首先分析写入操作的函数 ram_console_write()，具体实现代码如下所示。

```
static void ram_console_write(structconsole *console, const char *s, unsigned int count)
{
```

```
int rem;
//取得某缓冲区
struct ram_console_buffer *buffer =ram_console_buffer;
if (count >ram_console_buffer_size) {
s += count –ram_console_buffer_size;
count = ram_console_buffer_size;
}
rem = ram_console_buffer_size –buffer->start;
if (rem < count) {
ram_console_update(s, rem);
s += rem;
count -= rem;
buffer->start = 0;
buffer->size =ram_console_buffer_size;
}
ram_console_update(s, count);
buffer->start += count;
if (buffer->size <ram_console_buffer_size)
buffer->size += count;
ram_console_update_header();
}
```

在上述实现代码中，首先获取缓冲区 ram_console_buffer 的数值，然后判断要写入的最小数据量，并对缓冲区进行调整更新。如果定义了 CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION，则表示基于 RAM 的缓冲区，需要执行编码和解码操作。

写入更新操作功能由函数 ram_console_update()实现，具体实现代码如下所示。

```
static void ram_console_update(const char *s, unsigned int count)
{
struct ram_console_buffer *buffer = ram_console_buffer;
#ifdef CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION
uint8_t *buffer_end = buffer->data + ram_console_buffer_size;
uint8_t *block;
uint8_t *par;
int size = ECC_BLOCK_SIZE;
#endif
memcpy(buffer->data + buffer->start, s, count);
#ifdef CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION
block = buffer->data + (buffer->start & ~(ECC_BLOCK_SIZE - 1));
par = ram_console_par_buffer +
(buffer->start / ECC_BLOCK_SIZE) * ECC_SIZE;
do {
if (block + ECC_BLOCK_SIZE > buffer_end)
size = buffer_end - block;
ram_console_encode_rs8(block, size, par);
block += ECC_BLOCK_SIZE;
par += ECC_SIZE;
} while (block < buffer->data + buffer->start + count);
#endif
}
```

（9）接下来看编码操作和解码操作的具体实现，其中编码操作由函数 ram_console_encode_rs8()实现，具体实现代码如下所示。

```
static void ram_console_encode_rs8(uint8_t *data, size_t len, uint8_t *ecc)
{
  int i;
  uint16_t par[ECC_SIZE];
  /* Initialize the parity buffer */
  memset(par, 0, sizeof(par));
  encode_rs8(ram_console_rs_decoder, data, len, par, 0);
  for (i = 0; i < ECC_SIZE; i++)
  ecc[i] = par[i];
}
```

解码操作由函数 ram_console_decode_rs8()实现，具体实现代码如下所示。

```
static int ram_console_decode_rs8(void *data, size_t len, uint8_t *ecc)
{
int i;
uint16_t par[ECC_SIZE];
for (i = 0; i < ECC_SIZE; i++)
par[i] = ecc[i];
return decode_rs8(ram_console_rs_decoder, data, par, len,
NULL, 0, NULL, 0, NULL);
}
#endif
```

（10）函数 ram_console_update_header()的功能是更新信息头部标识，具体实现代码如下所示。

```
static void ram_console_update_header(void)
{
#ifdef CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION
struct ram_console_buffer *buffer = ram_console_buffer;
uint8_t *par;
par = ram_console_par_buffer +
DIV_ROUND_UP(ram_console_buffer_size, ECC_BLOCK_SIZE) * ECC_SIZE;
ram_console_encode_rs8((uint8_t *)buffer, sizeof(*buffer), par);
#endif
}
```

（11）函数 ram_console_save_old()的功能是将更新后的信息保存到 ram_console_old_log，具体实现代码如下所示。

```
static void __init
ram_console_save_old(struct ram_console_buffer *buffer, char *dest)
{
size_t old_log_size = buffer->size;
#ifdef CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION
uint8_t *block;
uint8_t *par;
char strbuf[80];
int strbuf_len;
block = buffer->data;
par = ram_console_par_buffer;
while (block < buffer->data + buffer->size) {
int numerr;
```

```
int size = ECC_BLOCK_SIZE;
if (block + size > buffer->data + ram_console_buffer_size)
size = buffer->data + ram_console_buffer_size - block;
numerr = ram_console_decode_rs8(block, size, par);
if (numerr > 0) {
#if 0
printk(KERN_INFO "ram_console: error in block %p, %d\n",
block, numerr);
#endif
ram_console_corrected_bytes += numerr;
} else if (numerr < 0) {
#if 0
printk(KERN_INFO "ram_console: uncorrectable error in "
"block %p\n", block);
#endif
ram_console_bad_blocks++;
}
block += ECC_BLOCK_SIZE;
par += ECC_SIZE;
}
if (ram_console_corrected_bytes || ram_console_bad_blocks)
strbuf_len = snprintf(strbuf, sizeof(strbuf),
"\n%d Corrected bytes, %d unrecoverable blocks\n",
ram_console_corrected_bytes, ram_console_bad_blocks);
else
strbuf_len = snprintf(strbuf, sizeof(strbuf),
"\nNo errors detected\n");
if (strbuf_len >= sizeof(strbuf))
strbuf_len = sizeof(strbuf) - 1;
old_log_size += strbuf_len;
#endif
if (dest == NULL) {
dest = kmalloc(old_log_size, GFP_KERNEL);
if (dest == NULL) {
printk(KERN_ERR
"ram_console: failed to allocate buffer\n");
return;
}
}
ram_console_old_log = dest;
ram_console_old_log_size = old_log_size;
memcpy(ram_console_old_log,
&buffer->data[buffer->start], buffer->size - buffer->start);
memcpy(ram_console_old_log + buffer->size - buffer->start,
&buffer->data[0], buffer->start);
#ifdef CONFIG_ANDROID_RAM_CONSOLE_ERROR_CORRECTION
memcpy(ram_console_old_log + old_log_size - strbuf_len,
strbuf, strbuf_len);
#endif
}
```

到此为止，Ram Console 驱动程序的调用工作全部结束。

第 13 章　USB Gadget 驱动

USB Gadget 是 Android 系统中一个基于标准 Linux USB Gadget 驱动框架的设备驱动。在 Android 系统中，新增了 ADB Gadget 驱动来实现 USB 驱动功能。当使用 Gadget 驱动时，Android 将作为一个 USB 设备而提供一个 ADB 接口。本章将详细讲解 Android 系统中 USB Gadget 驱动的基本架构知识，为读者学习本书后面的知识打下基础。

13.1　分析 Linux 内核的 USB 驱动程序

在分析 Android 系统的 USB 驱动程序之前，需要先了解 Linux 内核系统中的 USB 驱动程序。在 Linux 内核系统中，USB 驱动程序在目录 drivers/usb/中实现。

本节将详细分析 Linux 内核系统中 USB 驱动程序的基本知识。

13.1.1　USB 设备基础

Linux 内核系统中提供了一个名为 USB core 的子系统，通过此系统可以处理大部分的复杂功能，我们讲解的 USB 设备就是驱动程序和 USB core 之间的接口。在 USB 设备组织结构中，从上到下分为设备（device）、配置（config）、接口（interface）和端点（endpoint）4 个层次，具体说明如下。

☑ 设备：通常具有一个或多个的配置。
☑ 配置：经常具有一个或多个的接口。
☑ 接口：通常具有一个或多个的设置。
☑ 端点：没有或具有一个以上的端点。

下面将详细讲解上述各个组织结构成员的基本知识。

（1）设备

在 Linux 内核系统中，在文件 include/linux/usb.h 中使用数据结构 struct usb_device 来描述整个 USB 设备，这里的设备是一个可以实现插入操作的 USB 设备，具体实现代码如下所示。

```
struct usb_device {
  int devnum;                                //设备号，是在 USB 总线的地址
  char devpath [16];                         //消息的设备 ID 字符串
  enum usb_device_state state;               //设备状态：有已配置、未连接等状态
  enum usb_device_speed speed;               //设备速度：分为高速、全速、低速或错误
  struct usb_tt *tt;                         //处理传输者信息，分别处理低速、全速设备和高速 HUB
  int ttport;                                //位于 tt HUB 的设备口
  unsigned int toggle[2];                    //每个端点的占一位，表明端点的方向([0] = IN, [1] = OUT)
  struct usb_device *parent;                 //上一级 HUB 指针
  struct usb_bus *bus;                       //总线指针
  struct usb_host_endpoint ep0;              //端点 0 数据
  struct device dev;                         //一般的设备接口数据结构
  struct usb_device_descriptor descriptor;   //USB 设备描述符
```

```
    struct usb_host_config *config;                      //设备的所有配置
    struct usb_host_config *actconfig;                   //被激活的设备配置
    struct usb_host_endpoint *ep_in[16];                 //输入端点数组
    struct usb_host_endpoint *ep_out[16];                //输出端点数组
    char **rawdescriptors;                               //每个配置的 raw 描述符
    unsigned short bus_mA;                               //可使用的总线电流
    u8 portnum;                                          //父端口号
    u8 level;                                            //USB HUB 的层数

    unsigned can_submit:1;                               //URB 可被提交标志
    unsigned discon_suspended:1;                         //暂停时断开标志
    unsigned persist_enabled:1;                          //USB_PERSIST 使能标志
    unsigned have_langid:1;                              //string_langid 存在标志
    unsigned authorized:1;
    unsigned authenticated:1;
    unsigned wusb:1;                                     //无线 USB 标志

    int string_langid;                                   //字符串语言 ID
    /* static strings from the device */                 //设备的静态字符串
    char *product;                                       //产品名
    char *manufacturer;                                  //厂商名
    char *serial;                                        //产品串号
    struct list_head filelist;                           //此设备打开的 usbfs 文件
  #ifdef CONFIG_USB_DEVICE_CLASS
    struct device *usb_classdev;                         //用户空间访问的为 usbfs 设备创建的 USB 类设备
  #endif
  #ifdef CONFIG_USB_DEVICEFS
    struct dentry *usbfs_dentry;                         //设备的 usbfs 入口
  #endif
    int maxchild;                                        //（若为 HUB）接口数
    struct usb_device *children[USB_MAXCHILDREN];        //连接在这个 HUB 上的子设备
    int pm_usage_cnt;                                    //自动挂起的使用计数
    u32 quirks;
    atomic_t urbnum;                                     //这个设备所提交的 URB 计数

    unsigned long active_duration;                       //激活后使用计时
  #ifdef CONFIG_PM                                       //电源管理相关
    struct delayed_work autosuspend;                     //自动挂起的延时
    struct work_struct autoresume;                       //（中断的）自动唤醒需求
    struct mutex pm_mutex;                               //PM 的互斥锁

  unsigned long last_busy;                               //最后使用的时间
  int autosuspend_delay;
  unsigned long connect_time;                            //第一次连接的时间

  unsigned auto_pm:1;                                    //自动挂起/唤醒
  unsigned do_remote_wakeup:1;                           //远程唤醒
  unsigned reset_resume:1;                               //使用复位替代唤醒
  unsigned autosuspend_disabled:1;                       //挂起关闭
  unsigned autoresume_disabled:1;                        //唤醒关闭
  unsigned skip_sys_resume:1;                            //跳过下个系统唤醒
```

```
#endif
 struct wusb_dev *wusb_dev;                          //（如果为无线 USB）连接到 WUSB 特定的数据结构
};
```

（2）配置

在 Linux 系统中，一个 USB 设备可以使用多个配置，并且可在各个配置之间进行转换以改变设备的不同状态。例如某个设备可以通过下载固件（firmware）的方式来改变设备的使用状态，那么 USB 设备就要使用切换配置来实现这个功能，在同一个时刻只能有一个配置可以被激活。在 Linux 内核系统中，使用文件 include/linux/usb.h 中的结构 struct usb_host_config 来描述 USB 配置，具体实现代码如下所示。

```
struct usb_host_config {
    struct usb_config_descriptor desc;                              //配置描述符
    char *string; /* 配置的字符串指针（如果存在） */
    struct usb_interface_assoc_descriptor *intf_assoc[USB_MAXIADS];  //配置的接口联合描述符链表
    struct usb_interface *interface[USB_MAXINTERFACES];             //接口描述符链表
    struct usb_interface_cache *intf_cache[USB_MAXINTERFACES];
    unsigned char *extra; /* 额外的描述符 */
    int extralen;
};
```

在现实应用中，程序员编写的 USB 设备驱动通常不需要读写这些结构的任何值。

（3）接口

通常 USB 的端点被设置为接口，USB 接口只会处理一种 USB 逻辑连接。在现实应用中，一个 USB 接口代表一个基本功能，每个 USB 驱动控制一个接口。对于一个物理硬件设备来说，可能需要一个以上的驱动程序。例如在 Windows XP 系统中插入一个 USB 设备后，有时系统会识别出多个设备，并要求安装相应的多个驱动。

在 Linux 内核系统中，使用文件 include/linux/usb.h 中的结构 struct usb_interface 来描述 USB 接口，具体实现代码如下所示。

```
struct usb_interface {
/* 包含所有可用于该接口的可选设置的接口结构数组*/
/*每个 struct usb_host_interface 包含一套端点配置，即 struct usb_host_endpoint 结构所定义的端点配置，这些
接口结构没有特别的顺序*/
    struct usb_host_interface *altsetting;
    struct usb_host_interface *cur_altsetting; /*指向 altsetting 内部指针，表示当前激活的接口配置*/
    unsigned num_altsetting; /* 可选设置的数量*/
    struct usb_interface_assoc_descriptor *intf_assoc;
    /* 如果绑定到这个接口的 USB 驱动使用 USB 主设备号,这个变量包含由 USB 核心分配给接口的次设备号,
这种情况只在成功地调用 usb_register_dev 后才有效*/
    int minor;
    /*以下的数据在我们写的驱动中基本不用考虑，系统会自动设置*/
    enum usb_interface_condition condition; /* state of binding */
    unsigned is_active:1; /* the interface is not suspended */
    unsigned sysfs_files_created:1; /* the sysfs attributes exist */
    unsigned ep_devs_created:1; /* endpoint "devices" exist */
    unsigned unregistering:1; /* unregistration is in progress */
    unsigned needs_remote_wakeup:1; /* driver requires remote wakeup */
    unsigned needs_altsetting0:1; /* switch to altsetting 0 is pending */
    unsigned needs_binding:1; /* needs delayed unbind/rebind */
    unsigned reset_running:1;

    struct device dev; /* 接口特定的设备信息 */
```

```
	struct device *usb_dev;
	int pm_usage_cnt; /* usage counter for autosuspend */
	struct work_struct reset_ws; /* for resets in atomic context */
};

struct usb_host_interface {
	struct usb_interface_descriptor desc; //接口描述符

	struct usb_host_endpoint *endpoint; /*  这个接口的所有端点结构体的联合数组*/
	char *string; /*  接口描述字符串  */
	unsigned char *extra; /*  额外的描述符  */
	int extralen;
};
```

通过上述实现代码可知，USB 核心将其传递给 USB 驱动，并由 USB 驱动负责后续的控制。

（4）端点

在现实应用中，实现 USB 通信的最基本形式是通过端点来实现，一个 USB 端点只能向一个方向传输数据（从主机到设备或者从设备到主机），可以将端点看作是一个单向的管道。

在 Linux 系统中，一个 USB 端点有 4 种不同的类型，分别对应着 4 种不同的数据传送方式。

- ☑ 控制 CONTROL：控制端点被用来控制对 USB 设备的不同部分访问。通常用作配置设备、获取设备信息、发送命令到设备或获取设备状态报告，这些端点通常较小。每个 USB 设备都有一个控制端点称为“端点 0”，USB 核心在插入时被用来配置设备。USB 协议保证总有足够的带宽留给控制端点传送数据到设备。
- ☑ 中断 INTERRUPT：每当 USB 主机向设备请求数据时，中断端点以固定的速率传送小量的数据。这是传送 USB 键盘数据和鼠标数据的主要方法，并且还用以传送数据到 USB 设备的方式来控制设备，此时不用来传送大量数据。USB 协议保证总有足够的带宽留给中断端点以传送数据到设备。
- ☑ 批量 BULK：表示批量端点用以传送大量数据，这些端点常比中断端点大得多。USB 协议不保证传输在特定时间范围内完成，如果总线上没有足够的空间来发送整个 BULK 包，则被分为多个包进行传输。这些端点普遍用于打印机、USB Mass Storage 和 USB 网络设备上。
- ☑ 等时 ISOCHRONOUS：此类型端点也能批量传送大量数据，但是这个数据不能保证被送达。这些端点用在可以处理数据丢失的设备中，并且更多依赖于保持持续的数据流，如音频和视频设备等。

在 Linux 内核系统中，使用文件 include/linux/usb.h 中的结构 struct usb_host_endpoint 来描述端点，具体实现代码如下所示。

```
struct usb_host_endpoint {
	struct usb_endpoint_descriptor desc;		//端点描述符
	struct list_head urb_list;			//此端点的 urb 队列，由 USB 核心维护
	void *hcpriv;
	struct ep_device *ep_dev; /* For sysfs info */
	unsigned char *extra; /* Extra descriptors */
	int extralen;
	int enabled;
};
```

通过上述实现代码可知，结构 struct usb_host_endpoint 所包含的真实端点信息由另一个结构 struct usb_endpoint_descriptor 所描述，此结构体表示端点描述符，包含了所有的 USB 特定数据，具体实现代码如下所示。

```
struct usb_endpoint_descriptor {
	__u8 bLength;
```

```
        __u8 bDescriptorType;
/*这个是特定端点的 USB 地址，这个 8 位数据包含端点的方向，结合位掩码 USB_DIR_OUT 和 USB_DIR_IN 使用，确定这个端点的数据方向*/
        __u8 bEndpointAddress;
//这是端点的类型，位掩码如下
        __u8 bmAttributes;
/*端点可以一次处理的最大字节数。驱动可以发送比这个值大的数据量到端点，但是当真正传送到设备时，数据会被分为 wMaxPakcetSize 大小的块，对于高速设备，通过使用高位部分几个额外位，可用来支持端点的高带宽模式*/
        __le16 wMaxPacketSize;
//如果端点是中断类型，该值是端点的间隔设置，即端点的中断请求间的间隔时间，以毫秒为单位
        __u8 bInterval;
        /* NOTE: these two are _only_ in audio endpoints. */
        /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
        __u8 bRefresh;
        __u8 bSynchAddress;
} __attribute__ ((packed));
#define USB_DT_ENDPOINT_SIZE 7
#define USB_DT_ENDPOINT_AUDIO_SIZE 9 /* Audio extension */
/*
 * Endpoints
 */
#define USB_ENDPOINT_NUMBER_MASK 0x0f /* in bEndpointAddress 端点的 USB 地址掩码 */
#define USB_ENDPOINT_DIR_MASK 0x80 /* in bEndpointAddress 数据方向掩码 */
#define USB_DIR_OUT 0 /* to device */
#define USB_DIR_IN 0x80 /* to host */

#define USB_ENDPOINT_XFERTYPE_MASK 0x03 /* bmAttributes 的位掩码*/
#define USB_ENDPOINT_XFER_CONTROL 0
#define USB_ENDPOINT_XFER_ISOC 1
#define USB_ENDPOINT_XFER_BULK 2
#define USB_ENDPOINT_XFER_INT 3
#define USB_ENDPOINT_MAX_ADJUSTABLE 0x80
/*-------------------------------------------------------------------------*/
```

13.1.2　USB 和 sysfs

在 Linux 内核系统中，因为单个 USB 物理设备非常复杂，所以每一个设备在 sysfs 中的具体表示也非常复杂。物理 USB 设备（通过 struct usb_device 表示）和单个 USB 接口（由 struct usb_interface 表示）都作为单个设备在 sysfs 系统中出现，具体原因是因为这两个结构都包含一个名为 struct device 的结构。例如下面是一个 USB 鼠标在 sysfs 系统中的目录树。

```
/sys/devices/pci0000:00/0000:00:1a.0/usb3/3-1 //这里表示 usb_device 结构
.
|-- 3-1:1.0 //这里表示鼠标所对应的 usb_interface
| |-- 0003:046D:C018.0003
| | |-- driver -> ../../../../../../../bus/hid/drivers/generic-usb
| | |-- power
| | | `-- wakeup
| | |-- subsystem -> ../../../../../../../bus/hid
```

```
| | `-- uevent
| |-- bAlternateSetting
| |-- bInterfaceClass
| |-- bInterfaceNumber
| |-- bInterfaceProtocol
| |-- bInterfaceSubClass
| |-- bNumEndpoints
| |-- driver -> ../../../../../../bus/usb/drivers/usbhid
| |-- ep_81 -> usb_endpoint/usbdev3.4_ep81
| |-- input
| | `-- input6
| | |-- capabilities
| | | |-- abs
| | | |-- ev
| | | |-- ff
| | | |-- key
| | | |-- led
| | | |-- msc
| | | |-- rel
| | | |-- snd
| | | `-- sw
| | |-- device -> ../../../3-1:1.0
| | |-- event3
| | | |-- dev
| | | |-- device -> ../../input6
| | | |-- power
| | | | `-- wakeup
| | | |-- subsystem -> ../../../../../../../../../../class/input
| | | `-- uevent
| | |-- id
| | | |-- bustype
| | | |-- product
| | | |-- vendor
| | | `-- version
| | |-- modalias
| | |-- mouse1
| | | |-- dev
| | | |-- device -> ../../input6
| | | |-- power
| | | | `-- wakeup
| | | |-- subsystem -> ../../../../../../../../../../class/input
| | | `-- uevent
| | |-- name
| | |-- phys
| | |-- power
| | | `-- wakeup
| | |-- subsystem -> ../../../../../../../../../class/input
| | |-- uevent
| | `-- uniq
| |-- modalias
| |-- power
```

```
| | `-- wakeup
| |-- subsystem -> ../../../../../../bus/usb
| |-- supports_autosuspend
| |-- uevent
| `-- usb_endpoint
| `-- usbdev3.4_ep81
| |-- bEndpointAddress
| |-- bInterval
| |-- bLength
| |-- bmAttributes
| |-- dev
| |-- device -> ../../../3-1:1.0
| |-- direction
| |-- interval
| |-- power
| | `-- wakeup
| |-- subsystem -> ../../../../../../../../../class/usb_endpoint
| |-- type
| |-- uevent
| `-- wMaxPacketSize
|-- authorized
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPacketSize0
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- busnum
|-- configuration
|-- descriptors
|-- dev
|-- devnum
|-- driver -> ../../../../../bus/usb/drivers/usb
|-- ep_00 -> usb_endpoint/usbdev3.4_ep00
|-- idProduct
|-- idVendor
|-- manufacturer
|-- maxchild
|-- power
| |-- active_duration
| |-- autosuspend
| |-- connected_duration
| |-- level
| |-- persist
| `-- wakeup
|-- product
|-- quirks
```

```
|-- speed
|-- subsystem -> ../../../../../bus/usb
|-- uevent
|-- urbnum
|-- usb_endpoint
| `-- usbdev3.4_ep00
| |-- bEndpointAddress
| |-- bInterval
| |-- bLength
| |-- bmAttributes
| |-- dev
| |-- device -> ../../../3-1
| |-- direction
| |-- interval
| |-- power
| | `-- wakeup
| |-- subsystem -> ../../../../../../../class/usb_endpoint
| |-- type
| |-- uevent
| `-- wMaxPacketSize
`-- version

38 directories, 91 files
```

在 Linux 系统中规定，随着 USB 集线器层次的增加，集线器的端口号被添加到字符串中紧跟着链中之前的集线器的端口号后面。对于一个两层的树来说，其设备为 root_hub-hub_port-hub_port:config.interface，后面的依次类推。

13.1.3 urb 通信

在 Linux 系统中，USB 驱动程序和 USB 设备之间使用 urb 进行通信。urb 以异步传输的方式，同一个特定 USB 设备的特定端点实现数据发送和接收。USB 设备驱动不但可以根据驱动的需要来分配多个 urb 给一个端点，而且也可以重用单个 urb 给多个不同的端点。因为 USB 设备中的每个端点都可以处理一个 urb 队列，所以多个 urb 可在队列清空之前被发送到相同的端点。另外，urb 也可以在任何时间被提交它的驱动取消。如果设备被移除后，urb 也可以被 USB 核心取消。并且 urb 被动态创建并包含一个内部引用计数，使它们可以在最后一个用户释放它们时被自动释放。

在 Linux 内核系统中，一个典型 urb 的运作流程如下。

（1）首先创建一个 urb。

（2）将创建的 urb 分配给一个特定 USB 设备的特定端点。

（3）将 urb 提交给 USB 核心。

（4）urb 被 USB 核心提交给特定设备的特定 USB 主机控制器驱动。

（5）urb 被 USB 主机控制器驱动处理，并被传送到设备。

经过以上操作步骤之后，USB 主机控制器驱动会通知 USB 设备驱动可以进行通信了。

1．urb 描述

在 Linux 内核系统中，使用文件 include/linux/usb.h 中的结构 struct urb 来描述 urb，具体实现代码如下所示。

```
struct urb {
    /* private: usb core and host controller only fields in the urb */
```

```
    struct kref kref;                    /* urb 引用计数 */
    void *hcpriv;                        /* host 控制器的私有数据 */
    atomic_t use_count;                  /* 当前提交计数 */
    atomic_t reject;                     /* 提交失败计数 */
    int unlinked;                        /* 连接失败代码 */

    /* public: documented fields in the urb that can be used by drivers */
    struct list_head urb_list;           /* list head for use by the urb's
                             * current owner */
    struct list_head anchor_list;        /* the URB may be anchored */
    struct usb_anchor *anchor;
/* 指向这个 urb 要发送的目标 struct usb_device 的指针 */
/*这个变量必须在 urb 被发送到 USB 核心之前被 USB 驱动初始化*/
    struct usb_device *dev;
    struct usb_host_endpoint *ep;    /* (internal) pointer to endpoint */
/* 这个 urb 所要发送到的特定 struct usb_device 的端点消息 */
/*这个变量必须在 urb 被发送到 USB 核心之前被 USB 驱动初始化。必须由下面的函数生成*/
    unsigned int pipe;
/* 当 urb 开始由 USB 核心处理或处理结束时，这个变量被设置为 urb 的当前状态 */
/*USB 驱动可安全访问这个变量的唯一时间是在 urb 结束处理例程函数中。这个限制是为防止静态 */
/* 对于等时 urb，在这个变量中成功值(0)只表示这个 urb 是否已被去链*/
/*为获得等时 urb 的详细状态，应当检查 iso_frame_desc 变量 */
    int status;
    unsigned int transfer_flags;         /* 传输设置*/
/* 指向用于发送数据到设备（OUT urb）或者从设备接收数据（IN urb）的缓冲区指针*/
/*为了主机控制器驱动正确访问这个缓冲，必须使用 kmalloc 调用来创建，不是在堆栈或者静态内存中 */
/*对控制端点，这个缓冲区用于数据中转*/
    void *transfer_buffer;
    dma_addr_t transfer_dma;             /* 用于以 DMA 方式传送数据至 USB 设备的缓冲区*/
/* transfer_buffer 或者 transfer_dma 变量指向的缓冲区大小*/
/*如果这是 0，传送缓冲没有被 USB 核心所使用*/
/*对于一个 OUT 端点，如果这个端点大小比这个变量指定的值小 */
/* 对这个 USB 设备的传输将被分成更小的块，以正确地传送数据。这种大的传送以连续的 USB 帧进行*/
/*在一个 urb 中提交一个大块数据，并且使 USB 主机控制器划分为更小的块 */
/*比连续地顺序发送小缓冲的速度快得多*/
    int transfer_buffer_length;
/* 当这个 urb 完成后，该变量被设置为 urb（对于 OUT urb）发送或（对于 IN urb）接收数据的真实长度 */
/*对于 IN urb，必须是用此变量而非 transfer_buffer_length，因为接收的数据可能比整个缓冲小*/
    int actual_length;
/* 指向控制 urb 的设置数据包指针。它在传送缓冲中的数据之前被传送（用于控制 urb）*/
    unsigned char *setup_packet;
/* 控制 urb 用于设置数据包的 DMA 缓冲区地址，它在传送普通缓冲区中的数据之前被传送（用于控制 urb）*/
    dma_addr_t setup_dma;
    int start_frame;                     /* 设置或返回初始的帧数量（用于等时 urb） */
/* 指定 urb 所处理的等时传输缓冲区的数量（用于等时 urb，在 urb 被发送到 USB 核心前，必须设置） */
    int number_of_packets;
/*urb 被轮询的时间间隔，仅对中断或等时 urb 有效。这个值的单位依据设备速度而不同*/
/* 对于低速和高速的设备，单位是帧，它等同于毫秒。对于其他设备，单位是微帧，等同于 1/8 毫秒 */
/*在 urb 被发送到 USB 核心之前，此值必须设置*/
    int interval;
    int error_count;                     /* 等时 urb 的错误计数，由 USB 核心设置 */
```

```
/* 指向一个可以被 USB 驱动模块设置的数据块。当 urb 被返回到驱动时，可在结束处理例程中使用 */
	void *context;
/* 结束处理例程函数指针，当 urb 被完全传送或发生错误时，它将被 USB 核心调用 */
/* 此函数检查这个 urb，并决定释放它或重新提交给另一个传输中*/
	usb_complete_t complete;
/* （仅用于等时 urb）usb_iso_packet_descriptor 结构体允许单个 urb 一次定义许多等时传输*/
/*用于收集每个单独的传输状态*/
	struct usb_iso_packet_descriptor iso_frame_desc[0];

	};

struct usb_iso_packet_descriptor {
	unsigned int offset;		/* 该数据包的数据在传输缓冲区中的偏移量（第一个字节为 0） */
	unsigned int length;		/* 该数据包的传输缓冲区大小 */
	unsigned int actual_length;	/* 等时数据包接收到传输缓冲区中的数据长度 */
/* 该数据包的单个等时传输状态。它可以把相同的返回值作为主 struct urb 结构体的状态变量 */
	int status;
};

typedef void (*usb_complete_t)(struct urb *);
```

在上述结构体的实现代码中涉及生成函数 pipe()，此函数的具体实现代码如下所示。

```
static inline unsigned int __create_pipe(struct usb_device *dev,
		unsigned int endpoint)
{
	return (dev->devnum << 8) | (endpoint << 15);
}

/* Create various pipes... */
#define usb_sndctrlpipe(dev,endpoint)	\
	((PIPE_CONTROL << 30) | __create_pipe(dev, endpoint))
#define usb_rcvctrlpipe(dev,endpoint)	\
	((PIPE_CONTROL << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
#define usb_sndisocpipe(dev,endpoint)	\
	((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev, endpoint))
#define usb_rcvisocpipe(dev,endpoint)	\
	((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
#define usb_sndbulkpipe(dev,endpoint)	\
	((PIPE_BULK << 30) | __create_pipe(dev, endpoint))
#define usb_rcvbulkpipe(dev,endpoint)	\
	((PIPE_BULK << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
#define usb_sndintpipe(dev,endpoint)	\
	((PIPE_INTERRUPT << 30) | __create_pipe(dev, endpoint))
#define usb_rcvintpipe(dev,endpoint)	\
	((PIPE_INTERRUPT << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
//snd:OUT rcv:IN   ctrl:控制   isoc:等时   bulk:批量   int:中断
```

另外，结构 struct urb 中还使用 unsigned int transfer_flags 实现了传输设置，在里面设置了具体的值域，具体实现代码如下所示。

```
/* 当置位时，任何在 IN 端点上发生的简短读取，被 USB 核心当作错误。注意仅对从 USB 设备读取的 urb 有用 */
#define URB_SHORT_NOT_OK	0x0001
/* 如果为等时 urb，驱动想调度这个 urb 时，可置位该位，只要带宽允许且想在此时设置 urb 中的 start_frame 变量。若没有置位，则驱动必须指定 start_frame 值，且传输如果不能在当时启动的话，必须能够正确恢复 */
```

```
#define URB_ISO_ASAP           0x0002
/* 当 urb 包含要被发送的 DMA 缓冲时，应被置位。USB 核心就会使用 transfer_dma 变量指向的缓冲，而不是被
transfer_buffer 变量指向的缓冲 */
#define URB_NO_TRANSFER_DMA_MAP      0x0004
/*和 URB_NO_TRANSFER_DMA_MAP 类似，这个位用来控制 DMA 缓冲已经建立的 urb。如果它被置位，USB
核心使用 setup_dma 变量而不是 setup_packet 变量指向的缓冲 */
#define URB_NO_SETUP_DMA_MAP      0x0008
#define URB_NO_FSBR           0x0020
#define URB_ZERO_PACKET           0x0040
#define URB_NO_INTERRUPT      0x0080
#define URB_FREE_BUFFER           0x0100      /* Free transfer buffer with the URB */

#define URB_DIR_IN           0x0200      /* Transfer from device to host */
#define URB_DIR_OUT           0
#define URB_DIR_MASK           URB_DIR_IN
```

另外，结构体 int status 中的常用值在如下所示的文件中定义。

☑　include/asm-generic/errno.h

☑　include/asm-generic/errno_base.h

具体实现代码如下所示。

```
// 0       表示 urb 传送成功*/
//以下各个定义在使用时为负值
#define      ENOENT             2      /* urb 被 usb_kill_urb 停止 */
/* urb 被 usb_unlink_urb 去链，且 transfer_flags 被设为 URB_ASYNC_UNLINK */
#define      ECONNRESET      104
#define      EINPROGRESS      115/* urb 仍在 USB 主机控制器处理 */
#define      EPROTO           71      /* urb 发生错误：在传送中发生 bitstuff 错误或硬件没有及时收到响应帧 */
#define      EILSEQ           84      /* urb 传送中出现 CRC 校验错 */
/* 端点被停止。若此端点不是控制端点，则这个错误可通过函数 usb_clear_halt 清除 */
#define      EPIPE           32
#define      ECOMM           70      /* 数据传输时的接收速度快于写入系统内存的速度。此错误仅出现在 IN urb */
/* 从系统内存中获取数据的速度赶不上 USB 数据传送速度，此错误仅出现在 OUT urb */
#define      ENOSR           63
#define      EOVERFLOW      75 /* urb 发生"babble"（串扰）错误：端点接收的数据大于端点的最大数据包大小 */
/* 当 urb 的 transfer_flags 变量的 URB_SHORT_NOT_OK 标志被设置，urb 请求的数据没有完整地收到 */
#define      EREMOTEIO      181
#define      ENODEV           19      /* USB 设备从系统中拔出 */
/* 仅发生在等时 urb 中，表示传送部分完成。为了确定所传输的内容，驱动必须看单独的帧状态 */
#define      EXDEV           18
/* 如果 urb 的一个参数设置错误或在提交 urb 给 USB 核心的 usb_submit_urb 调用中 */
/*如果有不正确的参数则可能会发生此错误 */
#define      EINVAL           22
/* USB 主机控制器驱动有严重错误，它已被禁止或者设备从系统中拔出*/
/*并且这个 urb 在设备被移除后被提交。它也可能发生在 urb 被提交给设备时，设备的配置已被改变*/
#define      ESHUTDOWN      108
```

2．创建 urb

在 Linux 系统中，不能静态创建 struct urb 结构，必须使用函数 usb_alloc_urb()来创建，此函数的原型如下所示。

```
struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags);
```

各个参数的具体说明如下。

☑ int iso_packets：表示 urb 包含等时数据包的数目。如果不使用等时 urb，则为 0。

☑ gfp_t mem_flags：和传递给函数 kmalloc()调用从内核分配内存的标志类型相同。

如果驱动已经对 urb 使用完毕，则必须调用函数 usb_free_urb，此函数的原型如下所示。

```
void usb_free_urb(struct urb *urb);
```

参数 struct urb*urb 表示要释放的 struct urb 指针。

初始化 urb 的具体实现代码如下所示。

```
static inline void usb_fill_int_urb(struct urb *urb,
                        struct usb_device *dev,
                        unsigned int pipe,
                        void *transfer_buffer,
                        int buffer_length,
                        usb_complete_t complete_fn,
                        void *context,
                        int interval);

static inline void usb_fill_bulk_urb(struct urb *urb,
                        struct usb_device *dev,
                        unsigned int pipe,
                        void *transfer_buffer,
                        int buffer_length,
                        usb_complete_t complete_fn,
                        void *context);

static inline void usb_fill_control_urb(struct urb *urb,
                          struct usb_device *dev,
                          unsigned int pipe,
                          unsigned char *setup_packet,
                          void *transfer_buffer,
                          int buffer_length,
                          usb_complete_t complete_fn,
                          void *context);
urb->dev = dev;
urb->context = uvd;
urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp-1);
urb->interval = 1;
urb->transfer_flags = URB_ISO_ASAP;
urb->transfer_buffer = cam->sts_buf[i];
urb->complete = konicawc_isoc_irq;
urb->number_of_packets = FRAMES_PER_DESC;
urb->transfer_buffer_length = FRAMES_PER_DESC;
for (j=0; j < FRAMES_PER_DESC; j++) {
        urb->iso_frame_desc[j].offset = j;
        urb->iso_frame_desc[j].length = 1;
}
```

各个参数的具体说明如下。

☑ struct urb*urb：指向要被初始化的 urb 的指针。

☑ struct usb_device*dev：指向 urb 要发送到的 USB 设备。

- ☑ unsigned int pipe：urb 要被发送到的 USB 设备的特定端点。必须使用 usb_******pipe 格式的函数来创建。
- ☑ void*transfer_buffer：指向外发数据或接收数据的缓冲区的指针，不能是静态缓冲，必须使用 kmalloc 来创建。
- ☑ int buffer_length：transfer_buffer 指针指向的缓冲区的大小。
- ☑ usb_complete_t complete：指向 urb 结束处理例程函数指针。
- ☑ void*context：指向一个小数据块的指针，被添加到 urb 结构中，以便被结束处理例程函数获取使用。
- ☑ int interval：中断 urb 被调度的间隔。

3．提交 urb

在 Linux 系统中，当 urb 被正确地创建并初始化后就可以提交给 USB 核心以发送到 USB 设备。此功能是通过调用函数 usb_submit_urb()实现的，具体原型如下所示。

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags);
```

当 urb 被成功提交给 USB 核心之后，一直到结束处理例程函数被调用前都不能访问 urb 结构的任何成员。

4．注销 urb

在 Linux 系统中，使用如下所示的函数可以停止一个已经提交给 USB 核心的 urb。

```
void usb_kill_urb(struct urb *urb)
int usb_unlink_urb(struct urb *urb);
```

在现实应用中，调用函数 usb_kill_urb 会终止 urb 的生命周期，这通常在设备从系统移除时，在断开回调函数（disconnect callback）中被调用。

13.2　USB Gadget 驱动架构详解

和标准 Linux 内核系统相比，Android 系统专有的 USB 驱动程序是 USB Gadget，在 drivers/usb/gadget/目录中实现，本节将详细分析 Android 系统专有 USB 驱动程序 USB Gadget 的具体实现流程。

13.2.1　分析软件结构

在 Android 系统中，可以将整个 USB Gadget 驱动系统分为 3 层，分别是 UDC 层、USB 设备层和 Gadget 功能驱动层。下面将详细讲解这 3 层的基本知识。

1．UDC 层

在 USB Gadget 驱动系统中，UDC 层是与硬件相关层，与之相关的实现文件如下所示。

- ☑ drivers/usb/gadget/s3c2410_udc.c
- ☑ drivers/usb/gadget/s3c2410_udc.h

其中设备控制器 s3c2410 作为一个 Linux 设备，在 UDC 层作为 Platform 设备而注册到 Linux 设备模型中。接下来将详细讲解在 UDC 层中相关的数据结构和函数。

（1）数据结构

首先看数据结构 s3c2410_udc，具体实现代码如下所示。

```
struct s3c2410_udc {
	spinlock_t		lock;
```

```
	struct s3c2410_ep		ep[S3C2410_ENDPOINTS];
	int			address;
	struct usb_gadget		gadget;
	struct usb_gadget_driver	*driver;
	struct s3c2410_request		fifo_req;
	u8			fifo_buf[EP_FIFO_SIZE];
	u16			devstatus;

	u32			port_status;
	int			ep0state;

	unsigned		got_irq : 1;

	unsigned		req_std : 1;
	unsigned		req_config : 1;
	unsigned		req_pending : 1;
	u8			vbus;
	struct dentry		*regs_info;
};
```

在文件 s3c2410_udc.c 中声明了结构体变量 memory，通过此变量表示 S3C2410 的 USB 设备控制器，在其中包括了各种相关信息。结构体 memory 的定义代码如下所示。

```
static struct s3c2410_udc memory = {
	.gadget = {
		.ops		= &s3c2410_ops,
		.ep0		= &memory.ep[0].ep,
		.name		= gadget_name,
		.dev = {
			.init_name= "gadget",
		},
	},

	/* control endpoint */
	.ep[0] = {
		.num		= 0,
		.ep = {
			.name		= ep0name,
			.ops		= &s3c2410_ep_ops,
			.maxpacket	= EP0_FIFO_SIZE,
		},
		.dev		= &memory,
	},

	/* first group of endpoints */
	.ep[1] = {
		.num		= 1,
		.ep = {
			.name		= "ep1-bulk",
			.ops		= &s3c2410_ep_ops,
			.maxpacket	= EP_FIFO_SIZE,
		},
```

```
		.dev		= &memory,
		.fifo_size	= EP_FIFO_SIZE,
		.bEndpointAddress = 1,
		.bmAttributes	= USB_ENDPOINT_XFER_BULK,
	},
	.ep[2] = {
		.num		= 2,
		.ep = {
			.name		= "ep2-bulk",
			.ops		= &s3c2410_ep_ops,
			.maxpacket	= EP_FIFO_SIZE,
		},
		.dev		= &memory,
		.fifo_size	= EP_FIFO_SIZE,
		.bEndpointAddress = 2,
		.bmAttributes	= USB_ENDPOINT_XFER_BULK,
	},
	.ep[3] = {
		.num		= 3,
		.ep = {
			.name		= "ep3-bulk",
			.ops		= &s3c2410_ep_ops,
			.maxpacket	= EP_FIFO_SIZE,
		},
		.dev		= &memory,
		.fifo_size	= EP_FIFO_SIZE,
		.bEndpointAddress = 3,
		.bmAttributes	= USB_ENDPOINT_XFER_BULK,
	},
	.ep[4] = {
		.num		= 4,
		.ep = {
			.name		= "ep4-bulk",
			.ops		= &s3c2410_ep_ops,
			.maxpacket	= EP_FIFO_SIZE,
		},
		.dev		= &memory,
		.fifo_size	= EP_FIFO_SIZE,
		.bEndpointAddress = 4,
		.bmAttributes	= USB_ENDPOINT_XFER_BULK,
	}

};
```

再看 usb_gadget 操作函数集合，具体实现代码如下所示。

```
static const struct usb_gadget_ops s3c2410_ops = {
	.get_frame		= s3c2410_udc_get_frame,
	.wakeup			= s3c2410_udc_wakeup,
	.set_selfpowered	= s3c2410_udc_set_selfpowered,
	.pullup			= s3c2410_udc_pullup,
	.vbus_session		= s3c2410_udc_vbus_session,
```

```
	.vbus_draw		= s3c2410_vbus_draw,
};
```

上述代码中的函数都是由 UDC 层来实现的。

再看端点操作函数集合，具体实现代码如下所示。

```
static const struct usb_ep_ops s3c2410_ep_ops = {
	.enable		= s3c2410_udc_ep_enable,
	.disable	= s3c2410_udc_ep_disable,

	.alloc_request	= s3c2410_udc_alloc_request,
	.free_request	= s3c2410_udc_free_request,

	.queue		= s3c2410_udc_queue,
	.dequeue	= s3c2410_udc_dequeue,

	.set_halt	= s3c2410_udc_set_halt,
};
```

（2）函数

在整个驱动系统中，Platform 设备需要注册一个 platform_driver 的结构体，实现此功能的函数代码如下所示。

```
static struct platform_driver udc_driver_2410 = {
	.driver		= {
		.name	= "s3c2410-usbgadget",
		.owner	= THIS_MODULE,
	},
	.probe		= s3c2410_udc_probe,
	.remove		= s3c2410_udc_remove,
	.suspend	= s3c2410_udc_suspend,
	.resume		= s3c2410_udc_resume,
};
```

在结构体中的相关函数需要自己实现，其中最重要的函数是 s3c2410_udc_probe()，此函数在 platform 总线为驱动程序找到合适的设备后调用，在函数内初始化设备的时钟，申请 I/O 资源以及 irq 资源初始化 platform 设备结构体 struct s3c2410_udc memory。函数 s3c2410_udc_probe()的具体实现代码如下所示。

```
static int s3c2410_udc_probe(struct platform_device *pdev)
{
	struct s3c2410_udc *udc = &memory;
	struct device *dev = &pdev->dev;
	int retval;
	int irq;

	dev_dbg(dev, "%s()\n", __func__);

	usb_bus_clock = clk_get(NULL, "usb-bus-gadget");
	if (IS_ERR(usb_bus_clock)) {
		dev_err(dev, "failed to get usb bus clock source\n");
		return PTR_ERR(usb_bus_clock);
	}

	clk_enable(usb_bus_clock);
```

```

	udc_clock = clk_get(NULL, "usb-device");
	if (IS_ERR(udc_clock)) {
		dev_err(dev, "failed to get udc clock source\n");
		return PTR_ERR(udc_clock);
	}

	clk_enable(udc_clock);

	mdelay(10);

	dev_dbg(dev, "got and enabled clocks\n");

	if (strncmp(pdev->name, "s3c2440", 7) == 0) {
		dev_info(dev, "S3C2440: increasing FIFO to 128 bytes\n");
		memory.ep[1].fifo_size = S3C2440_EP_FIFO_SIZE;
		memory.ep[2].fifo_size = S3C2440_EP_FIFO_SIZE;
		memory.ep[3].fifo_size = S3C2440_EP_FIFO_SIZE;
		memory.ep[4].fifo_size = S3C2440_EP_FIFO_SIZE;
	}

	spin_lock_init(&udc->lock);
	udc_info = dev_get_platdata(&pdev->dev);

	rsrc_start = S3C2410_PA_USBDEV;
	rsrc_len   = S3C24XX_SZ_USBDEV;

	if (!request_mem_region(rsrc_start, rsrc_len, gadget_name))
		return -EBUSY;

	base_addr = ioremap(rsrc_start, rsrc_len);
	if (!base_addr) {
		retval = -ENOMEM;
		goto err_mem;
	}

	the_controller = udc;
	platform_set_drvdata(pdev, udc);

	s3c2410_udc_disable(udc);
	s3c2410_udc_reinit(udc);

	/* irq setup after old hardware state is cleaned up */
	retval = request_irq(IRQ_USBD, s3c2410_udc_irq,
				0, gadget_name, udc);

	if (retval != 0) {
		dev_err(dev, "cannot get irq %i, err %d\n", IRQ_USBD, retval);
		retval = -EBUSY;
		goto err_map;
```

```
	}

	dev_dbg(dev, "got irq %i\n", IRQ_USBD);

	if (udc_info && udc_info->vbus_pin > 0) {
		retval = gpio_request(udc_info->vbus_pin, "udc vbus");
		if (retval < 0) {
			dev_err(dev, "cannot claim vbus pin\n");
			goto err_int;
		}

		irq = gpio_to_irq(udc_info->vbus_pin);
		if (irq < 0) {
			dev_err(dev, "no irq for gpio vbus pin\n");
			retval = irq;
			goto err_gpio_claim;
		}

		retval = request_irq(irq, s3c2410_udc_vbus_irq,
					IRQF_TRIGGER_RISING
					| IRQF_TRIGGER_FALLING | IRQF_SHARED,
					gadget_name, udc);

		if (retval != 0) {
			dev_err(dev, "can't get vbus irq %d, err %d\n",
				irq, retval);
			retval = -EBUSY;
			goto err_gpio_claim;
		}

		dev_dbg(dev, "got irq %i\n", irq);
	} else {
		udc->vbus = 1;
	}

	if (udc_info && !udc_info->udc_command &&
		gpio_is_valid(udc_info->pullup_pin)) {

		retval = gpio_request_one(udc_info->pullup_pin,
				udc_info->vbus_pin_inverted ?
				GPIOF_OUT_INIT_HIGH : GPIOF_OUT_INIT_LOW,
				"udc pullup");
		if (retval)
			goto err_vbus_irq;
	}

	retval = usb_add_gadget_udc(&pdev->dev, &udc->gadget);
	if (retval)
		goto err_add_udc;

	if (s3c2410_udc_debugfs_root) {
```

```
                udc->regs_info = debugfs_create_file("registers", S_IRUGO,
                                s3c2410_udc_debugfs_root,
                                udc, &s3c2410_udc_debugfs_fops);
                if (!udc->regs_info)
                        dev_warn(dev, "debugfs file creation failed\n");
        }

        dev_dbg(dev, "probe ok\n");

        return 0;

err_add_udc:
        if (udc_info && !udc_info->udc_command &&
                        gpio_is_valid(udc_info->pullup_pin))
                gpio_free(udc_info->pullup_pin);
err_vbus_irq:
        if (udc_info && udc_info->vbus_pin > 0)
                free_irq(gpio_to_irq(udc_info->vbus_pin), udc);
err_gpio_claim:
        if (udc_info && udc_info->vbus_pin > 0)
                gpio_free(udc_info->vbus_pin);
err_int:
        free_irq(IRQ_USBD, udc);
err_map:
        iounmap(base_addr);
err_mem:
        release_mem_region(rsrc_start, rsrc_len);

        return retval;
}
```

到此为止，UDC 层中的数据结构以及函数介绍完毕，可以看出不同的 UDC 采取不同的策略。因为 s3c2410 是集成的 USB 设备控制器，所以是采用 Platform 驱动形式来注册的。如果系统是外接的 USB 设备控制器，则会采用相应总线的注册形式，例如 PCI 等。Platform 驱动的目的是分配资源以及实现硬件的初级初始化处理，而对 USB 设备层和功能驱动层没有任何影响。

2．USB 设备层

UDC 层与 USB 设备层是通过使用两个结构体与两个函数进行交互的，其中结构体 struct usb_gadget 和 struct usb_gadget_driver 都是嵌入在 struct s3c2410_udc 结构中的，由不同软件层的代码初始化。

首先看结构体 struct usb_gadget，在定义 memory 时在 UDC 层中进行了初始化处理，而结构体 struct usb_gadget_driver 是在 USB 设备层中初始化的，它通过 usb_gadget_register_driver(struct usb_gadget_driver *driver)函数从 USB 设备层传过来然后赋值给 memory。函数 usb_gadget_register_driver(struct usb_gadget_driver *driver)是 UDC 层与 USB 设备层进行交互的函数，设备层通过调用此函数与 UDC 层联系在一起，此函数将 usb_gadget 与 usb_gadget_driver 联系在一起。

UDC 层的基本任务是向 USB 设备层提供 usb_gadget_register_driver(struct usb_gadget_driver *driver)，另外还需要提供为 usb_gadget 服务的相关函数，这些函数会通过 usb_gadget 传递给 USB 设备层。UDC 层还需要提供 USB 设备的中断处理程序，中断处理尤其重要。因为所有的 USB 传输都是由主机发起，是否有 USB 传输完全由 USB 中断判定，所以 USB 中断处理程序是整个软件架构的核心。

USB 设备层属于硬件无关层，本层相关的代码在文件 drivers/usb/gadget/composite.c 中实现。

USB 设备层的功能是隔离 Gadget 功能驱动和硬件相关层，使功能驱动直接与 USB 设备层交互，而无须考虑硬件的相关细节。另外 USB 设备层提供了 USB 设备的一些基本数据结构，不同的 Gadget 功能驱动可以共同调用。USB 设备层的作用很重要，如果没有这一层，则每一个功能驱动都需要实现自己的 USB 设备，这样会导致代码重用率过高的情形。USB 设备层向下会与 UDC 层进行交互，向上会与 Gadget 功能驱动层进行交互。

当 USB 设备层向下与 UDC 层进行交互时，主要方式是通过调用 usb_gadget_register_driver()函数实现的，此函数是 UDC 层函数，传递的参数是一个 usb_gadget_driver 的结构体，定义此结构体的代码如下所示。

```
struct usb_gadget_driver {
	char			*function;
	enum usb_device_speed	speed;
	int			(*bind)(struct usb_gadget *);
	void			(*unbind)(struct usb_gadget *);
	int			(*setup)(struct usb_gadget *,
					const struct usb_ctrlrequest *);
	void			(*disconnect)(struct usb_gadget *);
	void			(*suspend)(struct usb_gadget *);
	void			(*resume)(struct usb_gadget *);

	/* FIXME support safe rmmod */
	struct device_driver	driver;
};
```

在文件 composite.c 中声明了结构体变量 composite_driver，这个结构体变量就是传给函数 usb_gadget_register_driver()的参数。结构体变量 composite_driver 的具体实现代码如下所示。

```
static struct usb_gadget_driver composite_driver = {
	.speed		= USB_SPEED_HIGH,

	.bind		= composite_bind,
	.unbind		= __exit_p(composite_unbind),

	.setup		= composite_setup,
	.disconnect	= composite_disconnect,

	.suspend	= composite_suspend,
	.resume		= composite_resume,

	.driver	= {
		.owner		= THIS_MODULE,
	},
};
```

在上述代码中定义的所有函数都需要自己实现，这些函数大部分都拥有相同的参数 usb_gadget，由此可以看出这些函数是与 UDC 层相关的。

前面介绍的数据结构是与 UDC 进行交互的，而下面将要介绍的数据结构和函数是 USB 设备层与 Gadget 功能驱动层进行交互的。

（1）数据结构

首先看数据结构 usb_composite_dev，具体实现代码如下所示。

```
struct usb_composite_dev {
	struct usb_gadget		*gadget;
```

```
        struct usb_request          *req;
        unsigned                    bufsiz;

        struct usb_configuration    *config;

        /* private: */
        /* internals */
        struct usb_device_descriptor    desc;
        struct list_head            configs;
        struct usb_composite_driver     *driver;
        u8                          next_string_id;

        /* the gadget driver won't enable the data pullup
         * while the deactivation count is nonzero.
         */
        unsigned                    deactivations;

        /* protects at least deactivation count */
        spinlock_t                  lock;
};
```

通过上述实现代码可以看出，结构 usb_composite_dev 代表一个 USB 设备。在此结构体中有设备的描述符信息和配置信息，也有指向 usb_gadget 与 usb_composite_driver 的指针。结构 usb_composite_dev 内嵌在了 usb_gadget 中，是在函数 composite_bind()中实现分配与初始化操作的。

再看结构体 usb_composite_driver，此结构体代表一个 USB 设备驱动，是联系功能驱动的主要数据结构，由功能驱动层声明并初始化。结构体 usb_composite_driver 的具体实现代码如下所示。

```
struct usb_composite_driver {
        const char                      *name;
        const struct usb_device_descriptor  *dev;
        struct usb_gadget_strings       **strings;

        /* REVISIT:   bind() functions can be marked __init, which
         * makes trouble for section mismatch analysis.   See if
         * we can't restructure things to avoid mismatching...
         */

        int             (*bind)(struct usb_composite_dev *);
        int             (*unbind)(struct usb_composite_dev *);

        /* global suspend hooks */
        void            (*suspend)(struct usb_composite_dev *);
        void            (*resume)(struct usb_composite_dev *);
};
```

（2）函数

这一层的核心函数是 usb_composite_register()，此函数被 Gadget 功能驱动层所调用，功能是初始化 composite_driver，并调用了函数 usb_gadget_register_driver()。在调用 usb_gadget_register_driver()函数后，UDC 层将与 USB 设备层联系到一起。函数 usb_composite_register()的具体实现代码如下所示。

```
int __init usb_composite_register(struct usb_composite_driver *driver)
{
```

```
	if (!driver || !driver->dev || !driver->bind || composite)
		return -EINVAL;

	if (!driver->name)
		driver->name = "composite";
	composite_driver.function =  (char *) driver->name;
	composite_driver.driver.name = driver->name;
	composite = driver;

	return usb_gadget_register_driver(&composite_driver);
}
```

因为函数 usb_composite_register()是在功能驱动的模块初始化函数中被调用的，所以只要加载了功能驱动，3 个软件层就可以通过数据结构联系在一起。

3．Gadget 功能驱动层

在 USB Gadget 驱动系统中，Gadget 功能驱动层位于整个驱动软件结构的最上层，功能是实现 USB 设备的功能。此层通常与 Linux 内核的其他层有密切的联系，例如模拟 U 盘的 Gadget 应用就与文件系统层和块 I/O 层有着千丝万缕的联系。本书主要讲解 Gadget 的功能驱动 zero，这一层的实现文件是 drivers/usb/gadget/zero.c。

功能驱动 zero 是作为一个模块注册到内核的，此模块初始化函数是 init()，具体实现代码如下所示。

```
static int __init init(void)
{
	return usb_composite_register(&zero_driver);
}
```

在上述代码中调用了函数 usb_composite_register()，一旦调用此函数就会将 USB Gadget 驱动系统的 3 个软件层联系到一起。USB Gadget 驱动系统的参数是 zero_driver，是一个 usb_composite_driver 的结构体，定义此结构体的实现代码如下所示。

```
static struct usb_composite_driver zero_driver = {
	.name		= "zero",
	.dev		= &device_desc,
	.strings	= dev_strings,
	.bind		= zero_bind,
	.unbind		= zero_unbind,
	.suspend	= zero_suspend,
	.resume		= zero_resume,
};
```

功能驱动 zero 只要实现上述代码中定义的函数即可。到此为止，USB Gadget 驱动系统的结构介绍完毕。整个层次的具体结构如图 13-1 所示。

由此可见，在这 3 层中两层是与硬件无关的，分别是 Gadget 功能驱动层和 USB 设备层，而 UDC 层是与硬件相关的。每一层都提供一种关键的数据结构和函数与其他层交互，具体说明如下。

- ☑ Gadget 功能驱动层：最主要的结构是 struct usb_composite_driver，这个结构在这一层定义，并且实现结构中的各个函数。
- ☑ USB 设备层：最主要的数据结构是 struct usb_composite_dev 与 usb_gadget_driver。前一个代表一个 USB 设备，而后一个是 Gadget 驱动，与 UDC 层交互。
- ☑ UDC 层：最主要的数据结构是 struct usb_gadget，通常包含在其他结构体中。这个结构体代表了一个 USB 设备控制器的所有关于 USB 通信的信息。

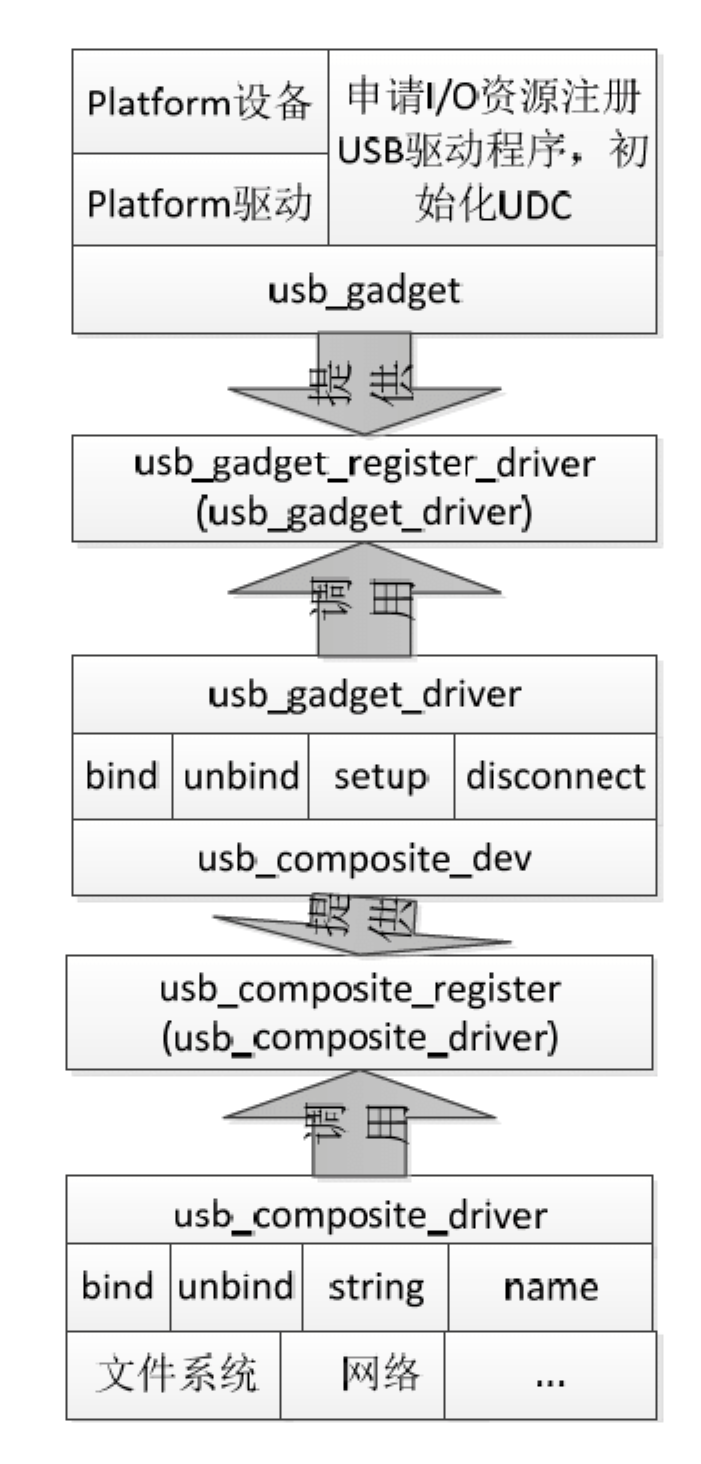


图 13-1　USB Gadget 驱动系统的结构

13.2.2　层次整合

在 USB Gadget 驱动系统中，UDC 层提供了由 USB 设备层调用的函数 usb_gadget_unregister_driver(struct usb_gadget_driver*driver)，USB 设备层将自己定义的结构变量 usb_gadget_driver 传递给整个函数。USB 设备层提供了由 Gadget 功能驱动层调用的函数 usb_composite_register(struct usb_composite_driver*driver)，Gadget 功能驱动层将自己定义的结构变量 usb_composite_driver 传递给整个函数。接下来将以 zero Gadget 功能驱动为例，将 s3c2410_udc 作为底层 UDC，详细分析这 3 层是如何整合在一起的。

（1）首先看 zero Gadget 功能驱动，它是作为一个模块注册到内核中的，此模块的初始化函数的实现代码如下所示。

```
static int __init init(void)
{
	return usb_composite_register(&zero_driver);
}
```

由此可见，初始化函数 init()只是调用了 usb_composite_register()函数，传递给它的参数是 zero_driver。此结构体的定义代码如下所示。

```
static struct usb_composite_driver zero_driver = {
	.name		= "zero",
	.dev		= &device_desc,
	.strings	= dev_strings,
	.bind		= zero_bind,
	.unbind		= zero_unbind,
	.suspend	= zero_suspend,
	.resume		= zero_resume,
};
```

在上述结构体中定义了多个函数，这些函数都是在文件 zero.c 中定义的。

（2）首先看函数 usb_composite_register()，此函数是由 USB 设备层提供的，在文件 composite.c 中定义，具体实现代码如下所示。

```
int __init usb_composite_register(struct usb_composite_driver *driver)
{
	if (!driver || !driver->dev || !driver->bind || composite)
		return -EINVAL;

	if (!driver->name)
		driver->name = "composite";
	composite_driver.function =  (char *) driver->name;
	composite_driver.driver.name = driver->name;
	composite = driver;

	return usb_gadget_register_driver(&composite_driver);
}
```

通过上述代码初始化了两个结构体变量，具体说明如下所示。

☑ 变量 composite_driver，这个变量是 USB 设备层定义的一个全局变量 usb_gadget_driver，具体实现代码如下所示。

```
static struct usb_gadget_driver composite_driver = {
	.speed		= USB_SPEED_HIGH,
	.bind		= composite_bind,
	.unbind		= __exit_p(composite_unbind),
	.setup		= composite_setup,
	.disconnect	= composite_disconnect,
	.suspend	= composite_suspend,
	.resume		= composite_resume,
	.driver	= {
		.owner		= THIS_MODULE,
	},
};
```

上述代码中定义的函数都是在 USB 设备层中实现的，其中函数 usb_composite_register()将 composite_driver 的 function 初始化为 zero，而 driver 是 struct device_driver 结构体，在 Linux 设备模型中使用，名字被初始化为 zero。

☑ 变量 composite，这是一个 USB 设备层定义的 usb_composite_driver()指针，这样 composite 就指向了 zero_driver，所以就将 zero Gadget()功能驱动层和 USB 设备层关联到了一起。

继续分析函数 usb_composite_register()，最后会调用函数 usb_gadget_register_driver()向 UDC 层联系。函数 usb_gadget_register_driver()在 UDC 层中定义，系统中的每个 UDC 都要实现这样一个函数。函数 usb_gadget_register_driver 的具体实现代码如下所示。

```
int usb_gadget_register_driver(struct usb_gadget_driver *driver)
{
//the_controller 指向已经初始化好了的 s3c2410_udc 结构，此结构代表了 s3c2410 usb 设备控制器，当然也包括
struct gadget 结构
	struct s3c2410_udc *udc = the_controller;
	int		retval;

	dprintk(DEBUG_NORMAL, "usb_gadget_register_driver() '%s'\n",
		driver->driver.name);
```

```
	/* Sanity checks */
	if (!udc)
		return -ENODEV;

	if (udc->driver)
		return -EBUSY;

	if (!driver->bind || !driver->setup
			|| driver->speed < USB_SPEED_FULL) {
		printk(KERN_ERR "Invalid driver: bind %p setup %p speed %d\n",
			driver->bind, driver->setup, driver->speed);
		return -EINVAL;
	}
#if defined(MODULE)
	if (!driver->unbind) {
		printk(KERN_ERR "Invalid driver: no unbind method\n");
		return -EINVAL;
	}
#endif
/*--------------上面的都是指针检查--------------*/
	//传递过来的 driver 就是 USB 设备层定义的 composite_driver，这样就成功联系了 UDC 层与 USB 设备层
	udc->driver = driver;
	//这里赋值的 driver 是 struct device_driver 结构，供 Linux 设备模型使用
	udc->gadget.dev.driver = &driver->driver;

	/*  绑定驱动  */
	if ((retval = device_add(&udc->gadget.dev)) != 0) {
		printk(KERN_ERR "Error in device_add() : %d\n",retval);
		goto register_error;
	}
	    //udc->gadget.dev 是 struct device  结构，这是向 Linux 设备模型核心注册设备

	dprintk(DEBUG_NORMAL, "binding gadget driver '%s'\n",
		driver->driver.name);

	if ((retval = driver->bind (&udc->gadget)) != 0) {
		device_del(&udc->gadget.dev);
		goto register_error;
	}

	/* Enable udc */
	s3c2410_udc_enable(udc);

	return 0;

register_error:
	udc->driver = NULL;
	udc->gadget.dev.driver = NULL;
	return retval;
}
```

在上述代码中，首先将 UDC 层与 USB 设备层关联在一起，然后调用函数 composite_bind()实现绑定工作。只有函数 usb_gadget_register_driver()执行完毕后这 3 层才可以真正地整合在一起，USB 设备才能正常工作。

（3）因为 Driver（驱动）就是传递过来的在 USB 设备层定义的 composite_driver，所以函数 composite_bind()是在 composite.c 中定义的，具体实现代码如下所示。

```
static int __init composite_bind(struct usb_gadget *gadget)
{
	struct usb_composite_dev*cdev;
	int			status = -ENOMEM;
//分配内存，struct usb_composite_dev 结构代表了一个 USB 设备
	cdev = kzalloc(sizeof *cdev, GFP_KERNEL);
	if (!cdev)
		return status;

	spin_lock_init(&cdev->lock);
	cdev->gadget = gadget;	//这个 gadget 是在文件 s3c2410_udc.c 中定义的
//下面的函数功能是使 gadget->dev->driver_data 指向 cdev 结构
//gadget->dev 是 struct device 结构已经注册到了 Linux 设备驱动模型核心
	set_gadget_data(gadget, cdev);
//cdev->configs 是 struct list_head 结构指针，这个链表将链接设备的所有配置
	INIT_LIST_HEAD(&cdev->configs);

	/* preallocate control response and buffer */
	cdev->req = usb_ep_alloc_request(gadget->ep0, GFP_KERNEL);
		if (!cdev->req)
		goto fail;
	cdev->req->buf = kmalloc(USB_BUFSIZ, GFP_KERNEL);
	if (!cdev->req->buf)
		goto fail;
	cdev->req->complete = composite_setup_complete;
	gadget->ep0->driver_data = cdev;

	cdev->bufsiz = USB_BUFSIZ;
//因为 struct usb_composite_dev 代表一个 USB 设备，所以它的驱动是 Gadget 功能驱动，此处是 composite，
//在前面 usb_composite_register 的时候赋值 zero_driver
	cdev->driver = composite;
//设置 USB 设备为自供电设备，因为设备 mini2440 开发板已经提供了电源，所以是自供电
	usb_gadget_set_selfpowered(gadget);
//此函数主要的功能是遍历 gadget 端点链表，将端点的 driver_data 清空
	usb_ep_autoconfig_reset(cdev->gadget);

//此函数调用涉及 Gadget 功能驱动层，也就是文件 zero.c
//composite->bind 定义在文件 zero.c 中，经过这个调用后，这 3 层才真正地联系在了一起
	status = composite->bind(cdev);
	if (status < 0)
		goto fail;
//以下代码都是设备描述符相关的
//cdev->desc 是 truct usb_device_descriptor 结构，代表了一个 USB 设备描述符
//这里用 Gadget 功能驱动层传递过来的参数初始化这个结构
	cdev->desc = *composite->dev;
	cdev->desc.bMaxPacketSize0 = gadget->ep0->maxpacket;
```

```
	/* standardized runtime overrides for device ID data */
	if (idVendor)
		cdev->desc.idVendor = cpu_to_le16(idVendor);
	if (idProduct)
		cdev->desc.idProduct = cpu_to_le16(idProduct);
	if (bcdDevice)
		cdev->desc.bcdDevice = cpu_to_le16(bcdDevice);

	/* strings can't be assigned before bind() allocates the
	 * releavnt identifiers
	 */
	if (cdev->desc.iManufacturer && iManufacturer)
		string_override(composite->strings,
			cdev->desc.iManufacturer, iManufacturer);
	if (cdev->desc.iProduct && iProduct)
		string_override(composite->strings,
			cdev->desc.iProduct, iProduct);
	if (cdev->desc.iSerialNumber && iSerialNumber)
		string_override(composite->strings,
			cdev->desc.iSerialNumber, iSerialNumber);

	INFO(cdev, "%s ready\n", composite->name);
	return 0;

fail:
	composite_unbind(gadget);
	return status;
}
```

在上述代码中，composite_bind 首先定义并初始化了结构体 usb_composite_dev，通过 cdev->gadget = gadget 语句将设备与底层的 gadget 联系在一起，通过 cdev->driver = composite 语句将设备与 Gadget 功能驱动联系在一起，并给设备端点 0 分配了一个 usb_request 结构（注意，这个结构在 USB 枚举将发挥重要的作用）。然后调用 Gadget 功能驱动层函数 bind()，最后初始化 USB 设备描述符。

（4）函数 composite_bind()的核心功能是调用 Gadget 功能驱动层的函数 bind()实现绑定，只有这样这 3 个软件层才能真正地联系在一起。在文件 zero.c 中定义了 zero Gadget 功能驱动层中的函数 bind()，具体实现代码如下所示。

```
static int __init zero_bind(struct usb_composite_dev *cdev)
{
	int			gcnum;
	struct usb_gadget	*gadget = cdev->gadget;
	int			id;

	/* Allocate string descriptor numbers ... note that string
	 * contents can be overridden by the composite_dev glue.
	 */
	id = usb_string_id(cdev);
//如果 cdev->next_string_id 不大于 254，将 cdev->next_string_id 加 1，
//返回加 1 后的 cdev->next_string_id。这里 cdev->next_string_id 为 0，所以执行完这个函数 id = 1;
	if (id < 0)
		return id;
```

```
	strings_dev[STRING_MANUFACTURER_IDX].id = id;
	device_desc.iManufacturer = id;
//strings_dev 是 zero 定义的字符串描述符数组，以上语句的作用是使生产厂商的字符串描述符的 ID 为 1
	id = usb_string_id(cdev);
	if (id < 0)
		return id;
	strings_dev[STRING_PRODUCT_IDX].id = id;
	device_desc.iProduct = id;
//以上语句的作用是使产品字符串描述符的 ID 为 2
	id = usb_string_id(cdev);
	if (id < 0)
		return id;
	strings_dev[STRING_SERIAL_IDX].id = id;
	device_desc.iSerialNumber = id;
//以上语句的作用是使生产串号的字符串描述符的 ID 为 3
	setup_timer(&autoresume_timer, zero_autoresume, (unsigned long) cdev);
//电源管理相关代码，暂时不用看
	/* Register primary, then secondary configuration.   Note that
	 * SH3 only allows one config...
	 */
	if (loopdefault) {
		loopback_add(cdev, autoresume != 0);
		if (!gadget_is_sh(gadget))
			sourcesink_add(cdev, autoresume != 0);
	} else {
		sourcesink_add(cdev, autoresume != 0);
		if (!gadget_is_sh(gadget))
			loopback_add(cdev, autoresume != 0);
	}
	gcnum = usb_gadget_controller_number(gadget);
	if (gcnum >= 0)
		device_desc.bcdDevice = cpu_to_le16(0x0200 + gcnum);
	else {
		pr_warning("%s: controller '%s' not recognized\n",
			longname, gadget->name);
		device_desc.bcdDevice = cpu_to_le16(0x9999);
	}


	INFO(cdev, "%s, version: " DRIVER_VERSION "\n", longname);

	snprintf(manufacturer, sizeof manufacturer, "%s %s with %s",
		init_utsname()->sysname, init_utsname()->release,
		gadget->name);

	return 0;
}
```

在上述实现代码中，首先设置了几个字符串描述符的 ID，然后设置 USB 配置。通过调用函数 sourcesink_add()传递了参数 cdev，也就是 USB 设备层定义的 USB 设备结构体。函数 sourcesink_add()在文件 f_sourcesink.c 中定义，具体实现代码如下所示。

```
int __init sourcesink_add(struct usb_composite_dev *cdev, bool autoresume)
{
	int id;

	/* 下面初始化一下字符串描述符的 ID */
	id = usb_string_id(cdev);
	if (id < 0)
		return id;
	strings_sourcesink[0].id = id;
	source_sink_intf.iInterface = id;
	sourcesink_driver.iConfiguration = id;
//source_sink_intf 是 struct usb_interface_descriptor 类型的变量，代表一个接口，sourcesink_driver 是 struct usb_configuration 类型的变量，代表一个 USB 配置，注意不是配置描述符。这两个变量在 f_sourcesink.c 中定义
	/* support autoresume for remote wakeup testing */
	if (autoresume)
		sourcesink_driver.bmAttributes |= USB_CONFIG_ATT_WAKEUP;

	/* support OTG systems */
	if (gadget_is_otg(cdev->gadget)) {
		sourcesink_driver.descriptors = otg_desc;
		sourcesink_driver.bmAttributes |= USB_CONFIG_ATT_WAKEUP;
	}

	return usb_add_config(cdev, &sourcesink_driver);
}
```

再看文件 f_sourcesink.c 中的数据结构 sourcesink_driver，此结构表示一个 USB 配置，在定义中说明了具体的配置功能。数据结构 sourcesink_driver 的定义代码如下所示。

```
static struct usb_configuration sourcesink_driver = {
	.label		= "source/sink",
	.strings	= sourcesink_strings,
	.bind		= sourcesink_bind_config,
	.setup		= sourcesink_setup,
	.bConfigurationValue = 3,
	.bmAttributes	= USB_CONFIG_ATT_SELFPOWER,
	/* .iConfiguration = DYNAMIC */
};
```

（5）在函数 sourcesink_add()中调用了函数 usb_add_config()，此函数传递了如下两个参数，分别是 USB 设备和 USB 配置。函数 usb_add_config()在文件 composite.c 中定义，功能是给 USB 设备增加一个配置，具体实现代码如下所示。

```
int __init usb_add_config(struct usb_composite_dev *cdev,
		struct usb_configuration *config)
{
	int				status = -EINVAL;
	struct usb_configuration	*c;

	DBG(cdev, "adding config #%u '%s'/%p\n",
			config->bConfigurationValue,
			config->label, config);

	if (!config->bConfigurationValue || !config->bind)
```

```
            goto done;

        /* Prevent duplicate configuration identifiers */
        list_for_each_entry(c, &cdev->configs, list) {
            if (c->bConfigurationValue == config->bConfigurationValue) {
                status = -EBUSY;
                goto done;
            }
        }
/*---上述代码用于检查参数的合法性---------*/
        config->cdev = cdev;
        list_add_tail(&config->list, &cdev->configs);
           //将配置加入到设备的配置链表中
        INIT_LIST_HEAD(&config->functions);
//初始化配置的 functions 链表，functions 链表要链接 struct usb_function 类型的数据结构，这个数据结构也很重要，它代表一个 USB 接口
        config->next_interface_id = 0;

        status = config->bind(config);
//这里函数调用的是 sourcesink_bind_config，这个函数的功能就是初始化一个 struct usb_function 结构，并且将其加入到配置的 functions 链表
        if (status < 0) { //status 小于 0 说明上面的函数调用失败所以删除配置
            list_del(&config->list);
            config->cdev = NULL;
        } else {    //给配置增加接口成功
            unsigned  i;
                  //打印调试信息
            DBG(cdev, "cfg %d/%p speeds:%s%s\n",
                config->bConfigurationValue, config,
                config->highspeed ? " high" : "",
                config->fullspeed
                    ? (gadget_is_dualspeed(cdev->gadget)
                        ? " full"
                        : " full/low")
                    : "");
            //MAX_CONFIG_INTERFACES 最大接口数，定义在文件 composite.h 中，为 16。每个配置可以有 16 个接口，以下代码遍历这个配置的所有接口，打印调试信息
            for (i = 0; i < MAX_CONFIG_INTERFACES; i++) {
                struct usb_function    *f = config->interface[i];

                if (!f)
                    continue;
                DBG(cdev, "    interface %d = %s/%p\n",
                    i, f->name, f);
            }
        }
//此函数的主要作用就是清空 cdev->gadget 的所有端点的 driver_data
        usb_ep_autoconfig_reset(cdev->gadget);

done:
        if (status)
```

```
            DBG(cdev, "added config '%s'/%u --> %d\n", config->label,
                        config->bConfigurationValue, status);
      return status;
}
```

通过上述代码实现了配置的初始化操作，成功地将配置与设备联系在了一起，并且打印输出了一些调试信息。

（6）当设备有了配置信息后，接下来需要调用函数 sourcesink_bind_config()给配置信息添加接口。函数 sourcesink_bind_config()的具体实现代码如下所示。

```
static int __init sourcesink_bind_config(struct usb_configuration *c)
{
      struct f_sourcesink   *ss;
      int                   status;

      ss = kzalloc(sizeof *ss, GFP_KERNEL);
      if (!ss)
            return -ENOMEM;

      ss->function.name = "source/sink";
      ss->function.descriptors = fs_source_sink_descs;
      ss->function.bind = sourcesink_bind;
      ss->function.unbind = sourcesink_unbind;
      ss->function.set_alt = sourcesink_set_alt;
      ss->function.disable = sourcesink_disable;

      status = usb_add_function(c, &ss->function);
      if (status)
            kfree(ss);
      return status;
}
```

通过上述实现代码分配并初始化了一个 struct f_sourcesink 结构体，此结构体包含了代表接口的结构 usb_function，并且初始化了结构体 usb_function 的以下回调函数，在最后调用函数 usb_add_function()将接口添加到配置信息中。函数 usb_add_function()的具体实现代码如下所示。

```
int __init usb_add_function(struct usb_configuration *config,
            struct usb_function *function)
{
      int    value = -EINVAL;
          //打印调试信息
      DBG(config->cdev, "adding '%s'/%p to config '%s'/%p\n",
                  function->name, function,
                  config->label, config);
         //检查参数合法性
      if (!function->set_alt || !function->disable)
            goto done;
          //添加接口到配置
      function->config = config;
      list_add_tail(&function->list, &config->functions);

      //如果 function 定义了 bind 函数则调用它，此处 function 定义了 bind 函数 sourcesink_bind()，此函数进行一些初始化的工作
```

```
	if (function->bind) {
		value = function->bind(config, function);
		if (value < 0) {
			list_del(&function->list);
			function->config = NULL;
		}
	} else
		value = 0;

	if (!config->fullspeed && function->descriptors)
		config->fullspeed = true;
	if (!config->highspeed && function->hs_descriptors)
		config->highspeed = true;

done:
	if (value)
		DBG(config->cdev, "adding '%s'/%p --> %d\n",
				function->name, function, value);
	return value;
}
```

（7）zero sourcesink 配置接口的 bind 为 sourcesink_bind，具体定义代码如下所示。

```
static int __init
sourcesink_bind(struct usb_configuration *c, struct usb_function *f)
{
	struct usb_composite_dev *cdev = c->cdev;
	struct f_sourcesink	*ss = func_to_ss(f);
	int	id;

	/* allocate interface ID(s) */
	id = usb_interface_id(c, f);
	if (id < 0)
		return id;
	source_sink_intf.bInterfaceNumber = id;
	  //usb_interface_id 的功能是判断 config->next_interface_id 是否大于 16，如果不是则执行 config->
interface[id] = f，再将 config->next_interface_id 加 1 返回
		ss->in_ep = usb_ep_autoconfig(cdev->gadget, &fs_source_desc);
	if (!ss->in_ep) {
autoconf_fail:
		ERROR(cdev, "%s: can't autoconfigure on %s\n",
			f->name, cdev->gadget->name);
		return -ENODEV;
	}
	ss->in_ep->driver_data = cdev; /* claim */

	ss->out_ep = usb_ep_autoconfig(cdev->gadget, &fs_sink_desc);
	if (!ss->out_ep)
		goto autoconf_fail;
	ss->out_ep->driver_data = cdev;	/* claim */

	/* support high speed hardware */
	if (gadget_is_dualspeed(c->cdev->gadget)) {
```

```
		hs_source_desc.bEndpointAddress =
				fs_source_desc.bEndpointAddress;
		hs_sink_desc.bEndpointAddress =
				fs_sink_desc.bEndpointAddress;
		f->hs_descriptors = hs_source_sink_descs;
	}

	DBG(cdev, "%s speed %s: IN/%s, OUT/%s\n",
			gadget_is_dualspeed(c->cdev->gadget) ? "dual" : "full",
			f->name, ss->in_ep->name, ss->out_ep->name);
	return 0;
}
```

到此为止，整个整合过程全部介绍完毕，成功地将 3 层整合在了一起。在整合成功后，这 3 层最终会形成一个饱满的 usb_composite_dev 结构，在此结构中包含了 USB 设备运行的各种信息，例如配置、接口和端点等。

通过上述实现过程可以看出，整合的过程大致分为如下两个过程。

（1）过程方向：Gadget 功能驱动层→USB 设备层→UDC 层。

此过程以如下 4 个数据结构为基础：

☑ usb_composite_driver
☑ usb_composite_dev
☑ usb_gadget_driver
☑ usb_gadget

此过程以如下两个 register 函数为导向：

☑ usb_composite_register(&zero_driver)
☑ usb_gadget_register_driver(&composite_driver)

（2）过程方向：UDC 层→USB 设备层→Gadget 功能驱动层。

此过程以 4 个绑定（bind）函数为点，引出了一连串数据结构与初始化过程，这 4 个绑定函数分别如下。

☑ USB 设备层的函数 composite_bind()：由 UDC 层的 usb_gadget_register_driver()函数调用，功能是分配 usb_composite_dev 并初始化。结构 usb_composite_dev 串联了 UDC 层的 usb_gadget 与 Gadget 功能驱动层的 usb_composite_driver，并且调用下一个上层的 bind。
☑ Gadget 功能驱动层的函数 zero_bind()：功能是用 Gadget 功能驱动层的 USB 设备信息来进一步初始化 usb_composite_dev 结构，并且引出下面两个 bind 函数。
☑ 函数 usb_add_config()：是一个与 USB 设备信息相关的 bind 函数，在添加配置时调用。
☑ 函数 usb_add_function()：是一个与 USB 设备信息相关的 bind 函数，在添加接口时调用。

13.2.3　USB 设备枚举

通过本章前面的内容可知，Gadget 功能驱动层、USB 设备层与 UDC 底层结合在一起，形成了一个完整的 USB 设备，而这个设备接下来需要接受主机的枚举，最终能够在主机中使用。

在 Linux 系统中，USB 设备枚举的基本步骤如下。

（1）将设备插入主机，主机检测到设备，复位设备。

（2）主机向设备控制端点发送 Get_Descriptor，以了解设备默认管道的大小。

（3）主机指定一个地址，发送 Set_Address 标准请求设置设备的地址。

（4）主机使用新的地址，再次发送 Get_Descriptor 以获得各种描述符。

（5）主机加载一个 USB 设备驱动。

（6）USB 设备驱动再发送 Set_Confuration 标准设备请求配置设备。

经过上述 6 个步骤的操作，USB 主机就可以识别我们的设备了。在上述设备枚举的过程中，USB 设备必须正确地响应主机的要求才能顺利完成设备枚举。下面将以 s3c2440 USB 设备控制器为例，详细分析上述枚举步骤的具体实现流程。本书为了节省篇幅，只是分析了 USB 设备枚举过程中的第二步：Get_Descriptor 阶段的控制传输。其他的步骤和此步骤相似，都是从主机端发起，然后 USB 设备通过终端来处理。

（1）当主机向 s3c2440 USB 设备控制器发送一个包时，USB 设备控制器就会产生相应的中断处理，并且当传输过程出现错误时也会以中断的形式进行通知。在文件 drivers/usb/gadget/s3c2410_udc.c 中，当实现设备初始化工作时就已经注册了中断处理程序。首先看函数 s3c2410_udc_irq()，功能是根据不同的中断类型进行处理，此函数的具体实现代码如下所示。

```
static irqreturn_t s3c2410_udc_irq(int dummy, void *_dev)
{
		struct s3c2410_udc *dev = _dev;
		int usb_status;
		int usbd_status;
		int pwr_reg;
		int ep0csr;
		int i;
		u32 idx, idx2;
		unsigned long flags;
		//自旋锁，用于保护 dev 结构避免并发引起的静态
		spin_lock_irqsave(&dev->lock, flags);

		/*当没有初始化好 USB 设备而发生中断时，清除中断标志*/
		if (!dev->driver) {
			/*  清除中断  */
			udc_write(udc_read(S3C2410_UDC_USB_INT_REG),
					S3C2410_UDC_USB_INT_REG);
			udc_write(udc_read(S3C2410_UDC_EP_INT_REG),
					S3C2410_UDC_EP_INT_REG);
		}

		/* s3c2440 USB 设备控制器，因为有 5 个端点，每个端点的寄存器都相似。所以硬件设计时将寄存
器分组了，名称一样但是物理寄存器不同。S3C2410_UDC_INDEX_REG 寄存器代表了哪个组*/
		idx = udc_read(S3C2410_UDC_INDEX_REG);

		/*读取状态寄存器的值至局部变量中*/
		usb_status = udc_read(S3C2410_UDC_USB_INT_REG);
		usbd_status = udc_read(S3C2410_UDC_EP_INT_REG);
		pwr_reg = udc_read(S3C2410_UDC_PWR_REG);

		udc_writeb(base_addr, S3C2410_UDC_INDEX_EP0, S3C2410_UDC_INDEX_REG);
		ep0csr = udc_read(S3C2410_UDC_IN_CSR1_REG);
		//输出调试信息
		dprintk(DEBUG_NORMAL, "usbs=%02x, usbds=%02x, pwr=%02x ep0csr=%02x\n",
			usb_status, usbd_status, pwr_reg, ep0csr);

		/*
		 * Now, handle interrupts. There's two types :
```

```
             * - Reset, Resume, Suspend coming -> usb_int_reg
             * - EP -> ep_int_reg
             */

        /*下面是不同的中断处理，复位对应设备的枚举值*/
        if (usb_status & S3C2410_UDC_USBINT_RESET) {
                /* two kind of reset :
                 * - reset start -> pwr reg = 8
                 * - reset end -> pwr reg = 0
                 **/
                dprintk(DEBUG_NORMAL, "USB reset csr %x pwr %x\n",
                        ep0csr, pwr_reg);

                dev->gadget.speed = USB_SPEED_UNKNOWN;
                udc_write(0x00, S3C2410_UDC_INDEX_REG);
                udc_write((dev->ep[0].ep.maxpacket & 0x7ff) >> 3,
                                S3C2410_UDC_MAXP_REG);
                dev->address = 0;

                dev->ep0state = EP0_IDLE;
                dev->gadget.speed = USB_SPEED_FULL;

                /* clear interrupt */
                udc_write(S3C2410_UDC_USBINT_RESET,
                                S3C2410_UDC_USB_INT_REG);

                udc_write(idx, S3C2410_UDC_INDEX_REG);
                spin_unlock_irqrestore(&dev->lock, flags);
                return IRQ_HANDLED;
        }

        /* RESUME */
        if (usb_status & S3C2410_UDC_USBINT_RESUME) {
                dprintk(DEBUG_NORMAL, "USB resume\n");

                /* clear interrupt */
                udc_write(S3C2410_UDC_USBINT_RESUME,
                                S3C2410_UDC_USB_INT_REG);

                if (dev->gadget.speed != USB_SPEED_UNKNOWN
                                && dev->driver
                                && dev->driver->resume)
                        dev->driver->resume(&dev->gadget);
        }

        /* SUSPEND */
        if (usb_status & S3C2410_UDC_USBINT_SUSPEND) {
                dprintk(DEBUG_NORMAL, "USB suspend\n");

                /* clear interrupt */
                udc_write(S3C2410_UDC_USBINT_SUSPEND,
```

```
                                        S3C2410_UDC_USB_INT_REG);

                if (dev->gadget.speed != USB_SPEED_UNKNOWN
                                        && dev->driver
                                        && dev->driver->suspend)
                        dev->driver->suspend(&dev->gadget);

                dev->ep0state = EP0_IDLE;
        }

        /* EP */
        /* control traffic */
        /* check on ep0csr != 0 is not a good idea as clearing in_pkt_ready
         * generate an interrupt
         */
        if (usbd_status & S3C2410_UDC_INT_EP0) {
                dprintk(DEBUG_VERBOSE, "USB ep0 irq\n");
                /* Clear the interrupt bit by setting it to 1 */
                udc_write(S3C2410_UDC_INT_EP0, S3C2410_UDC_EP_INT_REG);
                s3c2410_udc_handle_ep0(dev);
        }

        /* endpoint data transfers */
        for (i = 1; i < S3C2410_ENDPOINTS; i++) {
                u32 tmp = 1 << i;
                if (usbd_status & tmp) {
                        dprintk(DEBUG_VERBOSE, "USB ep%d irq\n", i);


                        /* Clear the interrupt bit by setting it to 1 */
                        udc_write(tmp, S3C2410_UDC_EP_INT_REG);
                        s3c2410_udc_handle_ep(&dev->ep[i]);
                }
        }

        /* what else causes this interrupt? a receive! who is it? */
        if (!usb_status && !usbd_status && !pwr_reg && !ep0csr) {
                for (i = 1; i < S3C2410_ENDPOINTS; i++) {
                        idx2 = udc_read(S3C2410_UDC_INDEX_REG);
                        udc_write(i, S3C2410_UDC_INDEX_REG);

                        if (udc_read(S3C2410_UDC_OUT_CSR1_REG) & 0x1)
                                s3c2410_udc_handle_ep(&dev->ep[i]);

                        /* restore index */
                        udc_write(idx2, S3C2410_UDC_INDEX_REG);
                }
        }

        dprintk(DEBUG_VERBOSE, "irq: %d s3c2410_udc_done.\n", IRQ_USBD);

```

```
        /* Restore old index */
        udc_write(idx, S3C2410_UDC_INDEX_REG);

        spin_unlock_irqrestore(&dev->lock, flags);

        return IRQ_HANDLED;
}
```

（2）因为在设备枚举的过程中，USB 设备控制器产生的都是端点为 0 的中断，所以需要调用函数 s3c2410_udc_handle_ep0()，此函数在文件 s3c2410_udc.c 中定义，具体实现流程如下。

☑ 首先判断 ep 的 queue 是否为空，这个字段链接了结构 s3c2410_request 的链表，结构 s3c2410_request 是对结构 usb_request 的简单封装，代表一个一次 USB 传输。如果不为空，则将获取的成员赋值给 req 变量。
☑ 然后读取端点 0 的状态寄存器 EP0_CSR，此寄存器描述了端点 0 的状态。
☑ 将端点 0 的状态读入到局部变量 ep0csr 中，在中断处理程序中按照 USB 设备枚举的过程，先中断复位，然后 USB 主机会发起一次控制传输来获得设备描述符。此控制传输操作是 Get_Descriptor 标准的设备请求。
☑ 当建立阶段完毕后，data 包的数据会写入端点 0 的 FIFO，设备控制器 s3c2410 USB 就会产生中断，对应的 EP0_CSR 的 SETUP_END 位会发生置位操作，此时可以判断这个状态。
☑ 调用相应的函数读取 FIFO 中的数据，然后针对不同的类型采取不同的操作（例如接收数据、发送数据操作）。

函数 s3c2410_udc_handle_ep0()的具体实现代码如下所示。

```
static void s3c2410_udc_handle_ep0(struct s3c2410_udc *dev)
{
        u32                             ep0csr;
        struct s3c2410_ep           *ep = &dev->ep[0];
        struct s3c2410_request   *req;
        struct usb_ctrlrequest    crq;

        if (list_empty(&ep->queue))
                req = NULL;
        else
                req = list_entry(ep->queue.next, struct s3c2410_request, queue);

        /* We make the assumption that S3C2410_UDC_IN_CSR1_REG equal to
         * S3C2410_UDC_EP0_CSR_REG when index is zero */

        udc_write(0, S3C2410_UDC_INDEX_REG);
        ep0csr = udc_read(S3C2410_UDC_IN_CSR1_REG);

        dprintk(DEBUG_NORMAL, "ep0csr %x ep0state %s\n",
                ep0csr, ep0states[dev->ep0state]);

        /* clear stall status */
        if (ep0csr & S3C2410_UDC_EP0_CSR_SENTSTL) {
                s3c2410_udc_nuke(dev, ep, -EPIPE);
                dprintk(DEBUG_NORMAL, "... clear SENT_STALL ...\n");
                s3c2410_udc_clear_ep0_sst(base_addr);
                dev->ep0state = EP0_IDLE;
```

```
			return;
		}

		/* clear setup end */
		if (ep0csr & S3C2410_UDC_EP0_CSR_SE) {
			dprintk(DEBUG_NORMAL, "... serviced SETUP_END ...\n");
			s3c2410_udc_nuke(dev, ep, 0);
			s3c2410_udc_clear_ep0_se(base_addr);
			dev->ep0state = EP0_IDLE;
		}

		switch (dev->ep0state) {
		case EP0_IDLE:
			s3c2410_udc_handle_ep0_idle(dev, ep, &crq, ep0csr);
			break;

		case EP0_IN_DATA_PHASE:			/* GET_DESCRIPTOR etc */
			dprintk(DEBUG_NORMAL, "EP0_IN_DATA_PHASE ... what now?\n");
			if (!(ep0csr & S3C2410_UDC_EP0_CSR_IPKRDY) && req)
				s3c2410_udc_write_fifo(ep, req);
			break;

		case EP0_OUT_DATA_PHASE:		/* SET_DESCRIPTOR etc */
			dprintk(DEBUG_NORMAL, "EP0_OUT_DATA_PHASE ... what now?\n");
			if ((ep0csr & S3C2410_UDC_EP0_CSR_OPKRDY) && req)
				s3c2410_udc_read_fifo(ep, req);
			break;

		case EP0_END_XFER:
			dprintk(DEBUG_NORMAL, "EP0_END_XFER ... what now?\n");
			dev->ep0state = EP0_IDLE;
			break;

		case EP0_STALL:
			dprintk(DEBUG_NORMAL, "EP0_STALL ... what now?\n");
			dev->ep0state = EP0_IDLE;
			break;
		}
}
```

在上述代码中用到了结构体 s3c2410_ep，此数据结构代表了一个 s3c2410 USB 设备控制器的一个端点，具体定义代码如下所示。

```
struct s3c2410_ep {
	struct list_head		queue;
	unsigned long			last_io;	/* jiffies timestamp */
	struct usb_gadget		*gadget;
	struct s3c2410_udc		*dev;
	const struct usb_endpoint_descriptor *desc;
	struct usb_ep			ep;
	u8				num;
```

```
	unsigned short		fifo_size;
	u8			bEndpointAddress;
	u8			bmAttributes;

	unsigned		halted : 1;
	unsigned		already_seen : 1;
	unsigned		setup_stage : 1;
};
```

（3）在函数 s3c2410_udc_handle_ep0()中调用了函数 s3c2410_udc_handle_ep0_idle()，功能是读取端点 0 FIFO 中的数据，此端点中的数据就是控制传输的类型。然后通过 case 语句判断到底是什么控制传输，并设置根据不同的控制传输类型执行不同的操作。函数 s3c2410_udc_handle_ep0_idle()的具体实现代码如下所示。

```
static void s3c2410_udc_handle_ep0_idle(struct s3c2410_udc *dev,
					struct s3c2410_ep *ep,
					struct usb_ctrlrequest *crq,
					u32 ep0csr)
{
	int len, ret, tmp;

	/* start control request? */
	if (!(ep0csr & S3C2410_UDC_EP0_CSR_OPKRDY))
		return;

	s3c2410_udc_nuke(dev, ep, -EPROTO);

	len = s3c2410_udc_read_fifo_crq(crq);
	if (len != sizeof(*crq)) {
		dprintk(DEBUG_NORMAL, "setup begin: fifo READ ERROR"
			" wanted %d bytes got %d. Stalling out...\n",
			sizeof(*crq), len);
		s3c2410_udc_set_ep0_ss(base_addr);
		return;
	}

	dprintk(DEBUG_NORMAL, "bRequest = %d bRequestType %d wLength = %d\n",
		crq->bRequest, crq->bRequestType, crq->wLength);

	/* cope with automagic for some standard requests. */
	dev->req_std = (crq->bRequestType & USB_TYPE_MASK)
		== USB_TYPE_STANDARD;
	dev->req_config = 0;
	dev->req_pending = 1;

	switch (crq->bRequest) {
	case USB_REQ_SET_CONFIGURATION:
		dprintk(DEBUG_NORMAL, "USB_REQ_SET_CONFIGURATION ...\n");

		if (crq->bRequestType == USB_RECIP_DEVICE) {
			dev->req_config = 1;
			s3c2410_udc_set_ep0_de_out(base_addr);
		}
```

```
			break;

		case USB_REQ_SET_INTERFACE:
			dprintk(DEBUG_NORMAL, "USB_REQ_SET_INTERFACE ...\n");

			if (crq->bRequestType == USB_RECIP_INTERFACE) {
				dev->req_config = 1;
				s3c2410_udc_set_ep0_de_out(base_addr);
			}
			break;

		case USB_REQ_SET_ADDRESS:
			dprintk(DEBUG_NORMAL, "USB_REQ_SET_ADDRESS ...\n");

			if (crq->bRequestType == USB_RECIP_DEVICE) {
				tmp = crq->wValue & 0x7F;
				dev->address = tmp;
				udc_write((tmp | S3C2410_UDC_FUNCADDR_UPDATE),
						S3C2410_UDC_FUNC_ADDR_REG);
				s3c2410_udc_set_ep0_de_out(base_addr);
				return;
			}
			break;

		case USB_REQ_GET_STATUS:
			dprintk(DEBUG_NORMAL, "USB_REQ_GET_STATUS ...\n");
			s3c2410_udc_clear_ep0_opr(base_addr);

			if (dev->req_std) {
				if (!s3c2410_udc_get_status(dev, crq))
					return;
			}
			break;

		case USB_REQ_CLEAR_FEATURE:
			s3c2410_udc_clear_ep0_opr(base_addr);

			if (crq->bRequestType != USB_RECIP_ENDPOINT)
				break;

			if (crq->wValue != USB_ENDPOINT_HALT || crq->wLength != 0)
				break;

			s3c2410_udc_set_halt(&dev->ep[crq->wIndex & 0x7f].ep, 0);
			s3c2410_udc_set_ep0_de_out(base_addr);
			return;

		case USB_REQ_SET_FEATURE:
			s3c2410_udc_clear_ep0_opr(base_addr);

			if (crq->bRequestType != USB_RECIP_ENDPOINT)
```

```
				break;

			if (crq->wValue != USB_ENDPOINT_HALT || crq->wLength != 0)
				break;

			s3c2410_udc_set_halt(&dev->ep[crq->wIndex & 0x7f].ep, 1);
			s3c2410_udc_set_ep0_de_out(base_addr);
			return;

		default:
			s3c2410_udc_clear_ep0_opr(base_addr);
			break;
		}

		if (crq->bRequestType & USB_DIR_IN)
			dev->ep0state = EP0_IN_DATA_PHASE;
		else
			dev->ep0state = EP0_OUT_DATA_PHASE;

		if (!dev->driver)
			return;

		/* deliver the request to the gadget driver */
		ret = dev->driver->setup(&dev->gadget, crq);
		if (ret < 0) {
			if (dev->req_config) {
				dprintk(DEBUG_NORMAL, "config change %02x fail %d?\n",
					crq->bRequest, ret);
				return;
			}

			if (ret == -EOPNOTSUPP)
				dprintk(DEBUG_NORMAL, "Operation not supported\n");
			else
				dprintk(DEBUG_NORMAL,
					"dev->driver->setup failed. (%d)\n", ret);

			udelay(5);
			s3c2410_udc_set_ep0_ss(base_addr);
			s3c2410_udc_set_ep0_de_out(base_addr);
			dev->ep0state = EP0_IDLE;
			/* deferred i/o == no response yet */
		} else if (dev->req_pending) {
			dprintk(DEBUG_VERBOSE, "dev->req_pending... what now?\n");
			dev->req_pending = 0;
		}

		dprintk(DEBUG_VERBOSE, "ep0state %s\n", ep0states[dev->ep0state]);
}
```

在上述代码中，dev->driver->setup 已经被函数 composite_setup()进行了类型初始化的操作。

（4）函数 composite_setup()在文件 drivers/usb/gadget/composite.c 中定义，功能是获取 USB 控制请求中的各个字段，然后初始化端点 0 的结构 usb_request，并设置完成回调函数 composite_setup_complete()的调用，通过 switch 语句来判断是何种控制传输。函数 composite_setup()的具体实现代码如下所示。

```
int
composite_setup(struct usb_gadget *gadget, const struct usb_ctrlrequest *ctrl)
{
	struct usb_composite_dev		*cdev = get_gadget_data(gadget);
	struct usb_request			*req = cdev->req;
	int					value = -EOPNOTSUPP;
	int					status = 0;
	u16					w_index = le16_to_cpu(ctrl->wIndex);
	u8					intf = w_index & 0xFF;
	u16					w_value = le16_to_cpu(ctrl->wValue);
	u16					w_length = le16_to_cpu(ctrl->wLength);
	struct usb_function			*f = NULL;
	u8					endp;

	/* partial re-init of the response message; the function or the
	 * gadget might need to intercept e.g. a control-OUT completion
	 * when we delegate to it
	 */
	req->zero = 0;
	req->complete = composite_setup_complete;
	req->length = 0;
	gadget->ep0->driver_data = cdev;

	switch (ctrl->bRequest) {

	/* we handle all standard USB descriptors */
	case USB_REQ_GET_DESCRIPTOR:
		if (ctrl->bRequestType != USB_DIR_IN)
			goto unknown;
		switch (w_value >> 8) {

		case USB_DT_DEVICE:
			cdev->desc.bNumConfigurations =
				count_configs(cdev, USB_DT_DEVICE);
			cdev->desc.bMaxPacketSize0 =
				cdev->gadget->ep0->maxpacket;
			if (gadget_is_superspeed(gadget)) {
				if (gadget->speed >= USB_SPEED_SUPER) {
					cdev->desc.bcdUSB = cpu_to_le16(0x0300);
					cdev->desc.bMaxPacketSize0 = 9;
				} else {
					cdev->desc.bcdUSB = cpu_to_le16(0x0210);
				}
			}

			value = min(w_length, (u16) sizeof cdev->desc);
			memcpy(req->buf, &cdev->desc, value);
```

```
					break;
			case USB_DT_DEVICE_QUALIFIER:
				if (!gadget_is_dualspeed(gadget) ||
				    gadget->speed >= USB_SPEED_SUPER)
					break;
				device_qual(cdev);
				value = min_t(int, w_length,
					sizeof(struct usb_qualifier_descriptor));
				break;
			case USB_DT_OTHER_SPEED_CONFIG:
				if (!gadget_is_dualspeed(gadget) ||
				    gadget->speed >= USB_SPEED_SUPER)
					break;
				/* FALLTHROUGH */
			case USB_DT_CONFIG:
				value = config_desc(cdev, w_value);
				if (value >= 0)
					value = min(w_length, (u16) value);
				break;
			case USB_DT_STRING:
				value = get_string(cdev, req->buf,
						w_index, w_value & 0xff);
				if (value >= 0)
					value = min(w_length, (u16) value);
				break;
			case USB_DT_BOS:
				if (gadget_is_superspeed(gadget)) {
					value = bos_desc(cdev);
					value = min(w_length, (u16) value);
				}
				break;
			}
			break;

		/* any number of configs can work */
		case USB_REQ_SET_CONFIGURATION:
			if (ctrl->bRequestType != 0)
				goto unknown;
			if (gadget_is_otg(gadget)) {
				if (gadget->a_hnp_support)
					DBG(cdev, "HNP available\n");
				else if (gadget->a_alt_hnp_support)
					DBG(cdev, "HNP on another port\n");
				else
					VDBG(cdev, "HNP inactive\n");
			}
			spin_lock(&cdev->lock);
			value = set_config(cdev, ctrl, w_value);
			spin_unlock(&cdev->lock);
			break;
		case USB_REQ_GET_CONFIGURATION:
```

```
			if (ctrl->bRequestType != USB_DIR_IN)
				goto unknown;
			if (cdev->config)
				*(u8 *)req->buf = cdev->config->bConfigurationValue;
			else
				*(u8 *)req->buf = 0;
			value = min(w_length, (u16) 1);
			break;

		/* function drivers must handle get/set altsetting; if there's
		 * no get() method, we know only altsetting zero works
		 */
		case USB_REQ_SET_INTERFACE:
			if (ctrl->bRequestType != USB_RECIP_INTERFACE)
				goto unknown;
			if (!cdev->config || intf >= MAX_CONFIG_INTERFACES)
				break;
			f = cdev->config->interface[intf];
			if (!f)
				break;
			if (w_value && !f->set_alt)
				break;
			value = f->set_alt(f, w_index, w_value);
			if (value == USB_GADGET_DELAYED_STATUS) {
				DBG(cdev,
				 "%s: interface %d (%s) requested delayed status\n",
						__func__, intf, f->name);
				cdev->delayed_status++;
				DBG(cdev, "delayed_status count %d\n",
						cdev->delayed_status);
			}
			break;
		case USB_REQ_GET_INTERFACE:
			if (ctrl->bRequestType != (USB_DIR_IN|USB_RECIP_INTERFACE))
				goto unknown;
			if (!cdev->config || intf >= MAX_CONFIG_INTERFACES)
				break;
			f = cdev->config->interface[intf];
			if (!f)
				break;
			/* lots of interfaces only need altsetting zero... */
			value = f->get_alt ? f->get_alt(f, w_index) : 0;
			if (value < 0)
				break;
			*((u8 *)req->buf) = value;
			value = min(w_length, (u16) 1);
			break;

		/*
		 * USB 3.0 additions:
		 * Function driver should handle get_status request. If such cb
```

```
	 * wasn't supplied we respond with default value = 0
	 * Note: function driver should supply such cb only for the first
	 * interface of the function
	 */
	case USB_REQ_GET_STATUS:
		if (!gadget_is_superspeed(gadget))
			goto unknown;
		if (ctrl->bRequestType != (USB_DIR_IN | USB_RECIP_INTERFACE))
			goto unknown;
		value = 2;	/* This is the length of the get_status reply */
		put_unaligned_le16(0, req->buf);
		if (!cdev->config || intf >= MAX_CONFIG_INTERFACES)
			break;
		f = cdev->config->interface[intf];
		if (!f)
			break;
		status = f->get_status ? f->get_status(f) : 0;
		if (status < 0)
			break;
		put_unaligned_le16(status & 0x0000ffff, req->buf);
		break;
	/*
	 * Function drivers should handle SetFeature/ClearFeature
	 * (FUNCTION_SUSPEND) request. function_suspend cb should be supplied
	 * only for the first interface of the function
	 */
	case USB_REQ_CLEAR_FEATURE:
	case USB_REQ_SET_FEATURE:
		if (!gadget_is_superspeed(gadget))
			goto unknown;
		if (ctrl->bRequestType != (USB_DIR_OUT | USB_RECIP_INTERFACE))
			goto unknown;
		switch (w_value) {
		case USB_INTRF_FUNC_SUSPEND:
			if (!cdev->config || intf >= MAX_CONFIG_INTERFACES)
				break;
			f = cdev->config->interface[intf];
			if (!f)
				break;
			value = 0;
			if (f->func_suspend)
				value = f->func_suspend(f, w_index >> 8);
			if (value < 0) {
				ERROR(cdev,
				      "func_suspend() returned error %d\n",
				      value);
				value = 0;
			}
			break;
		}
		break;
```

```
	default:
unknown:
		VDBG(cdev,
			"non-core control req%02x.%02x v%04x i%04x l%d\n",
			ctrl->bRequestType, ctrl->bRequest,
			w_value, w_index, w_length);

		/* functions always handle their interfaces and endpoints...
		 * punt other recipients (other, WUSB, ...) to the current
		 * configuration code.
		 *
		 * REVISIT it could make sense to let the composite device
		 * take such requests too, if that's ever needed:    to work
		 * in config 0, etc
		 */
		switch (ctrl->bRequestType & USB_RECIP_MASK) {
		case USB_RECIP_INTERFACE:
			if (!cdev->config || intf >= MAX_CONFIG_INTERFACES)
				break;
			f = cdev->config->interface[intf];
			break;

		case USB_RECIP_ENDPOINT:
			endp = ((w_index & 0x80) >> 3) | (w_index & 0x0f);
			list_for_each_entry(f, &cdev->config->functions, list) {
				if (test_bit(endp, f->endpoints))
					break;
			}
			if (&f->list == &cdev->config->functions)
				f = NULL;
			break;
		}

		if (f && f->setup)
			value = f->setup(f, ctrl);
		else {
			struct usb_configuration		*c;

			c = cdev->config;
			if (c && c->setup)
				value = c->setup(c, ctrl);
		}

		goto done;
	}

	/* respond with data transfer before status phase? */
	if (value >= 0 && value != USB_GADGET_DELAYED_STATUS) {
		req->length = value;
		req->zero = value < w_length;
		value = usb_ep_queue(gadget->ep0, req, GFP_ATOMIC);
```

```
                if (value < 0) {
                        DBG(cdev, "ep_queue --> %d\n", value);
                        req->status = 0;
                        composite_setup_complete(gadget->ep0, req);
                }
        } else if (value == USB_GADGET_DELAYED_STATUS && w_length != 0) {
                WARN(cdev,
                        "%s: Delayed status not supported for w_length != 0",
                        __func__);
        }

done:
        /* device either stalls (value < 0) or reports success */
        return value;
}
```

在上述代码中，因为是 Get_Descriptor 传输控制类型，而且在设备枚举时只获取了设备描述符的前 8 个字节以了解端点 0 的 FIFO 深度，所以会执行下面的代码复制设备描述符到 req 的缓冲区。

```
cdev->desc.bNumConfigurations =
        count_configs(cdev, USB_DT_DEVICE);
value = min(w_length, (u16) sizeof cdev->desc);
memcpy(req->buf, &cdev->desc, value);
```

（5）函数 usb_ep_queue()的功能是将 req 中的数据写入 FIFO 中，具体实现代码如下所示。

```
static int s3c2410_udc_queue(struct usb_ep *_ep, struct usb_request *_req,
                gfp_t gfp_flags)
{
        struct s3c2410_request  *req = to_s3c2410_req(_req);
        struct s3c2410_ep       *ep = to_s3c2410_ep(_ep);
        struct s3c2410_udc      *dev;
        u32                     ep_csr = 0;
        int                     fifo_count = 0;
        unsigned long           flags;

        if (unlikely (!_ep || (!ep->desc && ep->ep.name != ep0name))) {
                dprintk(DEBUG_NORMAL, "%s: invalid args\n", __func__);
                return -EINVAL;
        }

        dev = ep->dev;
        if (unlikely (!dev->driver
                        || dev->gadget.speed == USB_SPEED_UNKNOWN)) {
                return -ESHUTDOWN;
        }
//以上检查参数合法性
        local_irq_save (flags);

        if (unlikely(!_req || !_req->complete
                        || !_req->buf || !list_empty(&req->queue))) {
                if (!_req)
                        dprintk(DEBUG_NORMAL, "%s: 1 X X X\n", __func__);
                else {
```

```
                dprintk(DEBUG_NORMAL, "%s: 0 %01d %01d %01d\n",
                    __func__, !_req->complete,!_req->buf,
                    !list_empty(&req->queue));
            }

            local_irq_restore(flags);
            return -EINVAL;
        }

        _req->status = -EINPROGRESS;
        _req->actual = 0;
//表示传输正在处理中
        dprintk(DEBUG_VERBOSE, "%s: ep%x len %d\n",
                __func__, ep->bEndpointAddress, _req->length);
//针对普通端点，对于端点 0 执行 else 以后的语句
        if (ep->bEndpointAddress) {
            udc_write(ep->bEndpointAddress & 0x7F, S3C2410_UDC_INDEX_REG);
            ep_csr = udc_read((ep->bEndpointAddress & USB_DIR_IN)
                    ? S3C2410_UDC_IN_CSR1_REG
                    : S3C2410_UDC_OUT_CSR1_REG);

            fifo_count = s3c2410_udc_fifo_count_out();
        } else { //端点 0
            udc_write(0, S3C2410_UDC_INDEX_REG);
            ep_csr = udc_read(S3C2410_UDC_IN_CSR1_REG);
            fifo_count = s3c2410_udc_fifo_count_out(); //读出当前 fifo 的位置
        }

        //如果端点的 urt 链表为空而端点正常，则执行下面的语句
        if (list_empty(&ep->queue) && !ep->halted) {
            if (ep->bEndpointAddress == 0 /* ep0 */) {
                switch (dev->ep0state) {
//对于 Get_Descriptor，在 s3c2410_udc_handle_ep0_idle 中已经设置 dev->ep0state 为 EP0_IN_DATA_PHASE，
所以执行下面的代码
                case EP0_IN_DATA_PHASE:
                    if (!(ep_csr&S3C2410_UDC_EP0_CSR_IPKRDY)
                            && s3c2410_udc_write_fifo(ep,
                                req)) {
                        dev->ep0state = EP0_IDLE;
                        req = NULL;
                    }
                    break;
                case EP0_OUT_DATA_PHASE:
                    if ((!_req->length)
                        || ((ep_csr & S3C2410_UDC_OCSR1_PKTRDY)
                            && s3c2410_udc_read_fifo(ep,
                                req))) {
                        dev->ep0state = EP0_IDLE;
                        req = NULL;
                    }
                    break;
```

```
                default:
                        local_irq_restore(flags);
                        return -EL2HLT;
                }
        } else if ((ep->bEndpointAddress & USB_DIR_IN) != 0
                        && (!(ep_csr&S3C2410_UDC_OCSR1_PKTRDY))
                        && s3c2410_udc_write_fifo(ep, req)) {
                req = NULL;
        } else if ((ep_csr & S3C2410_UDC_OCSR1_PKTRDY)
                        && fifo_count
                        && s3c2410_udc_read_fifo(ep, req)) {
                req = NULL;
        }
    }

    /* pio or dma irq handler advances the queue. */
    if (likely (req != 0)) //如果 req 为 0，下面的代码不执行
            list_add_tail(&req->queue, &ep->queue);
    local_irq_restore(flags);

    dprintk(DEBUG_VERBOSE, "%s ok\n", __func__);
    return 0;
}
```

通过上述代码，整个处理过程又返回到了函数 composite_setup()中。到此为止，整个枚举过程的第二步 Get_Descriptor 阶段的控制传输工作全部结束。

13.3　实战演练——USB 驱动例程分析

在 Linux 系统中，文件 drivers/usb/usb-skeleton.c 是 Linux 内核为我们提供的最基础的 USB 驱动程序，是一个 USB 驱动开发的骨架程序。本节将以此文件作为基础，详细分析 USB 驱动的实现过程。

13.3.1　结构体 usb_device_id

在 Linux 系统中，驱动程序会把驱动设备对象注册到 USB 的子系统中，然后使用供应商（idVendor）和设备（idProduct）标识来判断是否已经安装了对应的硬件设备。其中通过结构体 usb_device_id 提供了这个驱动可以支持的不同类型 USB 设备的列表，USB 核心可以通过这个列表来决定设备对应的驱动。并且热插拔脚本可以通过此列表来决定当特定设备被插入系统时，应该自动加载哪一个驱动。结构体 usb_device_id 的具体实现代码如下所示。

```
struct usb_device_id {
    /* 确定设备信息和结构体中的哪几个字段匹配来判断驱动的适用性 */
    __u16           match_flags;
    /* Used for product specific matches; range is inclusive */
    __u16           idVendor;       //USB 设备的制造商 ID，需向 www.usb.org 申请
    __u16           idProduct;      //USB 设备的产品 ID，由制造商自定
    __u16           bcdDevice_lo;       /* USB 设备的产品版本号最低值*/
```

```
	__u16		bcdDevice_hi;	/* USB 设备的产品版本号最高值，以 BCD 码来表示*/

	/* 分别定义设备的类、子类和协议，它们由 USB 论坛分配并定义在 USB 规范中。这些值指定这个设备的行为，包括设备上所有的接口 */
	__u8		bDeviceClass;
	__u8		bDeviceSubClass;
	__u8		bDeviceProtocol;

	/* 分别定义单个接口的类、子类和协议，它们由 USB 论坛分配并定义在 USB 规范中 */
	__u8		bInterfaceClass;
	__u8		bInterfaceSubClass;
	__u8		bInterfaceProtocol;

	/* 这个值不用来匹配驱动，驱动用它在 USB 驱动的探测回调函数中区分不同的设备 */
	kernel_ulong_t	driver_info;
};
```

在上述结构体的实现代码中，通过__u16 match_flags 所使用的 define 来判断驱动的适应性，在文件 include/linux/mod_devicetable.h 中定义了具体的字段，具体实现代码如下所示。

```
/* Some useful macros to use to create struct usb_device_id */
#define USB_DEVICE_ID_MATCH_VENDOR		0x0001
#define USB_DEVICE_ID_MATCH_PRODUCT		0x0002
#define USB_DEVICE_ID_MATCH_DEV_LO		0x0004
#define USB_DEVICE_ID_MATCH_DEV_HI		0x0008
#define USB_DEVICE_ID_MATCH_DEV_CLASS		0x0010
#define USB_DEVICE_ID_MATCH_DEV_SUBCLASS	0x0020
#define USB_DEVICE_ID_MATCH_DEV_PROTOCOL	0x0040
#define USB_DEVICE_ID_MATCH_INT_CLASS		0x0080
#define USB_DEVICE_ID_MATCH_INT_SUBCLASS	0x0100
#define USB_DEVICE_ID_MATCH_INT_PROTOCOL	0x0200

//include/linux/usb.h
#define USB_DEVICE_ID_MATCH_DEVICE \
		(USB_DEVICE_ID_MATCH_VENDOR | USB_DEVICE_ID_MATCH_PRODUCT)
#define USB_DEVICE_ID_MATCH_DEV_RANGE \
		(USB_DEVICE_ID_MATCH_DEV_LO | USB_DEVICE_ID_MATCH_DEV_HI)
#define USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION \
		(USB_DEVICE_ID_MATCH_DEVICE | USB_DEVICE_ID_MATCH_DEV_RANGE)
#define USB_DEVICE_ID_MATCH_DEV_INFO \
		(USB_DEVICE_ID_MATCH_DEV_CLASS | \
		USB_DEVICE_ID_MATCH_DEV_SUBCLASS | \
		USB_DEVICE_ID_MATCH_DEV_PROTOCOL)
#define USB_DEVICE_ID_MATCH_INT_INFO \
		(USB_DEVICE_ID_MATCH_INT_CLASS | \
		USB_DEVICE_ID_MATCH_INT_SUBCLASS | \
		USB_DEVICE_ID_MATCH_INT_PROTOCOL)
//仅和指定的制造商和产品 ID 匹配，用于需要特定驱动的设备
#define USB_DEVICE(vend,prod) \
	.match_flags = USB_DEVICE_ID_MATCH_DEVICE, \
	.idVendor = (vend), \
	.idProduct = (prod)
```

```
//仅和某版本范围内的指定的制造商和产品 ID 匹配
#define USB_DEVICE_VER(vend, prod, lo, hi) \
	.match_flags = USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION, \
	.idVendor = (vend), \
	.idProduct = (prod), \
	.bcdDevice_lo = (lo), \
	.bcdDevice_hi = (hi)
//仅和指定的接口协议、制造商和产品 ID 匹配
#define USB_DEVICE_INTERFACE_PROTOCOL(vend, prod, pr) \
	.match_flags = USB_DEVICE_ID_MATCH_DEVICE | \
		USB_DEVICE_ID_MATCH_INT_PROTOCOL, \
	.idVendor = (vend), \
	.idProduct = (prod), \
	.bInterfaceProtocol = (pr)
//仅和指定的设备类型相匹配
#define USB_DEVICE_INFO(cl, sc, pr) \
	.match_flags = USB_DEVICE_ID_MATCH_DEV_INFO, \
	.bDeviceClass = (cl), \
	.bDeviceSubClass = (sc), \
	.bDeviceProtocol = (pr)
//仅和指定的接口类型相匹配
#define USB_INTERFACE_INFO(cl, sc, pr) \
	.match_flags = USB_DEVICE_ID_MATCH_INT_INFO, \
	.bInterfaceClass = (cl), \
	.bInterfaceSubClass = (sc), \
	.bInterfaceProtocol = (pr)
//仅和指定的制造商、产品 ID 和接口类型相匹配
#define USB_DEVICE_AND_INTERFACE_INFO(vend, prod, cl, sc, pr) \
	.match_flags = USB_DEVICE_ID_MATCH_INT_INFO \
		| USB_DEVICE_ID_MATCH_DEVICE, \
	.idVendor = (vend), \
	.idProduct = (prod), \
	.bInterfaceClass = (cl), \
	.bInterfaceSubClass = (sc), \
	.bInterfaceProtocol = (pr)
/* ------------------------------------------------------------------------- */
```

13.3.2　结构体 usb_driver

USB 骨架程序的 usb_driver 结构体的定义，具体实现代码如下所示。

```
static struct usb_driver skel_driver = {
	.name =			"skeleton",
	.probe =		skel_probe,
	.disconnect =	skel_disconnect,
	.suspend =		skel_suspend,
	.resume =		skel_resume,
	.pre_reset =	skel_pre_reset,
	.post_reset =	skel_post_reset,
	.id_table =		skel_table,
	.supports_autosuspend = 1,
};
```

对于一个只为一个供应商的一个 USB 设备服务的 USB 设备驱动来说，定义和初始化列表数组的代码如下所示。

```
static const struct usb_device_id skel_table[] = {
        { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
        { }                                    /* Terminating entry */
};
MODULE_DEVICE_TABLE(usb, skel_table);
```

在上述代码中，宏 MODULE_DEVICE_TABLE 是必需的，它允许用户空间工具判断该驱动可以控制什么设备。对于 USB 驱动来说，此宏中的第一个值必须是 usb。

如果需要编写的驱动被系统中每个 USB 设备所调用，在创建时只需设置 driver_info 成员即可，具体实现代码如下所示。

```
static struct usb_device_id usb_ids[] = {
 {.driver_info = 42},
    {}
};
```

13.3.3　注册 USB 驱动程序

在 Linux 系统中，所有 USB 驱动都必须创建结构体 usb_driver，此结构必须被 USB 驱动程序手动填充并包含多个回调函数和变量，并且需要向 USB 核心描述 USB 驱动程序。结构体 usb_driver 的具体实现代码如下所示。

```
struct usb_driver {
    const char *name;
    /*指向驱动程序名字的指针，它必须在内核所有的 USB 驱动中是唯一的（通常被设为和驱动模块名相同）。
当驱动在内核中运行时，会出现在/sys/bus/usb/drivers 目录中  */
    int (*probe) (struct usb_interface *intf,
          const struct usb_device_id *id);
    /*  指向 USB 驱动中探测函数指针  */
  /*当 USB 核心认为它有一个本驱动可处理的 struct usb_interface 时此函数将被调用  */
/*USB 核心用来作判断的 struct usb_device_id 指针也被传递给此函数  */
/*如果这个 USB 驱动确认传递给它的 struct usb_interface，则应当正确地初始化设备并返回 0 */
/*如果驱动没有确认这个设备或发生错误，则返回负错误值  */

    void (*disconnect) (struct usb_interface *intf);
    /*指向 USB 驱动的断开函数指针*/
/*当 struct usb_interface 从系统中清除或驱动 USB 核心卸载时，函数将被 USB 核心调用*/

    int (*ioctl) (struct usb_interface *intf, unsigned int code,
            void *buf);
    /*指向 USB 驱动的 ioctl 函数指针  */
/*若此函数存在，在用户空间程序对 usbfs 文件系统关联的设备调用 ioctl 时，此函数将被调用  */
/*实际上，当前只有 USB 集线器驱动使用这个 ioctl*/

    int (*suspend) (struct usb_interface *intf, pm_message_t message);
    /*指向 USB 驱动中挂起函数的指针*/
    int (*resume) (struct usb_interface *intf);
    /*指向 USB 驱动中恢复函数的指针*/
    int (*reset_resume)(struct usb_interface *intf);
```

```
	/*要复位一个已经被挂起的 USB 设备时调用此函数*/

	int (*pre_reset)(struct usb_interface *intf);
	/*在设备被复位之前由 usb_reset_composite_device()调用*/
	int (*post_reset)(struct usb_interface *intf);
	/*在设备被复位之后由 usb_reset_composite_device()调用*/

	const struct usb_device_id *id_table;
	/*指向 struct usb_device_id 表的指针*/

	struct usb_dynids dynids;
	struct usbdrv_wrap drvwrap;
	/*是 struct device_driver driver 的再包装，struct device_driver 包含 struct module *owner;*/
	unsigned int no_dynamic_id:1;
	unsigned int supports_autosuspend:1;
	unsigned int soft_unbind:1;
};
#define	to_usb_driver(d) container_of(d, struct usb_driver, drvwrap.driver)
```

在结构体 usb_driver 中，有如下两个 USB 核心在适当时调用的函数。

☑ 当设备安装时，如果 USB 核心认为这个驱动可以处理，则调用探测函数检查传递给它的设备信息，并判断这个驱动是否真正适合这个设备。

☑ 因为某些原因，设备被移除或驱动不再控制设备时，调用断开函数进行适当的清理工作。

13.3.4　加载和卸载 USB 骨架程序模块

实现 USB 骨架程序模块的加载和卸载功能，具体实现代码如下所示。

```
//向 USB 核心注册 struct usb_driver
static int __init usb_skel_init(void)
{
		int result;
		/* register this driver with the USB subsystem */
		result = usb_register(&skel_driver);
		if (result)
				err("usb_register failed. Error number %d", result);
		return result;
}

/*当 USB 驱动被卸载，struct usb_driver 需要从内核注销（代码如下）。当以下调用发生时，当前绑定到这个驱动的任何 USB 接口将会断开并调用断开函数*/
static void __exit usb_skel_exit(void)
{
		/* deregister this driver with the USB subsystem */
		usb_deregister(&skel_driver);
}
```

13.3.5　探测回调函数

Linux 系统中的 USB 驱动需要初始化可能用来管理 USB 设备的所有本地结构，并需要保存所有需要的设备

信息到本地结构。为了和设备进行通信，USB 驱动通常需要探测设备的端点地址和缓冲大小。文件 usb-skeleton 中的探测函数 skel_probe()的实现代码如下所示。

```
static int skel_probe(struct usb_interface *interface,
                      const struct usb_device_id *id)
{
        struct usb_skel *dev;
        struct usb_host_interface *iface_desc;
        struct usb_endpoint_descriptor *endpoint;
        size_t buffer_size;
        int i;
        int retval = -ENOMEM;
        /*  为结构体 usb_skel  分配内存空间*/
        dev = kzalloc(sizeof(*dev), GFP_KERNEL);
        if (!dev) {
                dev_err(&interface->dev, "Out of memory\n");
                goto error;
        }
        //开始初始化 usb_skel
        kref_init(&dev->kref);
        sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
        mutex_init(&dev->io_mutex);
        spin_lock_init(&dev->err_lock);
        init_usb_anchor(&dev->submitted);
        init_waitqueue_head(&dev->bulk_in_wait);

        dev->udev = usb_get_dev(interface_to_usbdev(interface));
        dev->interface = interface;

        /*设置终端信息  */
        /*  在此只是用第一个 bulk-in 和 bulk-out */
        iface_desc = interface->cur_altsetting;
        for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
                endpoint = &iface_desc->endpoint[i].desc;

                if (!dev->bulk_in_endpointAddr &&
                    usb_endpoint_is_bulk_in(endpoint)) {
                        /*  发现一个批量输入端点  */
                        buffer_size = usb_endpoint_maxp(endpoint);
                        dev->bulk_in_size = buffer_size;
                        dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
                        dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
                        if (!dev->bulk_in_buffer) {
                                dev_err(&interface->dev,
                                        "Could not allocate bulk_in_buffer\n");
                                goto error;
                        }
                        dev->bulk_in_urb = usb_alloc_urb(0, GFP_KERNEL);
                        if (!dev->bulk_in_urb) {
                                dev_err(&interface->dev,
                                        "Could not allocate bulk_in_urb\n");
```

```
                        goto error;
                    }
                }

                if (!dev->bulk_out_endpointAddr &&
                    usb_endpoint_is_bulk_out(endpoint)) {
                        /*发现一个批量输出端点 t */
                        dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
                }
        }
        if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
                dev_err(&interface->dev,
                        "Could not find both bulk-in and bulk-out endpoints\n");
                goto error;
        }

        /* 在接口设备中保存数据指针 */
        usb_set_intfdata(interface, dev);

        /* 注册 USB 设备*/
        retval = usb_register_dev(interface, &skel_class);
        if (retval) {
                /* something prevented us from registering this driver */
                dev_err(&interface->dev,
                        "Not able to get a minor for this device.\n");
                usb_set_intfdata(interface, NULL);
                goto error;
        }

        /* 通知用户设备依附于什么 node */
        dev_info(&interface->dev,
                 "USB Skeleton device now attached to USBSkel-%d",
                 interface->minor);
        return 0;

error:
        if (dev)
                /* 释放分配的内存*/
                kref_put(&dev->kref, skel_delete);
        return retval;
}
```

13.3.6　清理数据

函数 skel_disconnect()的功能是当设备被移除或驱动不再控制设备时，调用此函数做一些清理方面的工作。函数 skel_disconnect()的具体实现代码如下所示。

```
static void skel_disconnect(struct usb_interface *interface)
{
        struct usb_skel *dev;
        int minor = interface->minor;
```

```

        dev = usb_get_intfdata(interface);
        usb_set_intfdata(interface, NULL);

        /*  注销 USB 设备  */
        usb_deregister_dev(interface, &skel_class);

        /* prevent more I/O from starting */
        mutex_lock(&dev->io_mutex);
        dev->interface = NULL;
        mutex_unlock(&dev->io_mutex);

        usb_kill_anchored_urbs(&dev->submitted);

        /* decrement our usage count */
        kref_put(&dev->kref, skel_delete);

        dev_info(&interface->dev, "USB Skeleton #%d now disconnected", minor);
}
```

在上述代码中，调用函数 usb_get_intfdata()来获取端点的数据。当一个 USB 设备调用 disconnect()函数时，所有当前正被传送的 urb 可自动被 USB 核心取消，而无须显式调用 usb_kill_urb()。在 USB 设备被断开之后，如果驱动想调用函数 usb_submit_urb()去提交 urb 则会失败，错误值为-EPIPE。

13.3.7　函数 skel_write()和 skel_write_bulk_callback()

因为在中断上下文中运行 urb 回调函数，所以它不应做任何内存分配，持有任何信号量或任何可导致进程休眠的事情。如果从回调中提交 urb 并需要分配新的内存块，则需要使用标志 GFP_ATOMIC 来通知 USB 核心不要休眠。函数 skel_write()和 skel_write_bulk_callback()的具体实现代码如下所示。

```
static ssize_t skel_write(struct file *file, const char *user_buffer, size_t count, loff_t *ppos)
{
    struct usb_skel *dev;
    int retval = 0;
    struct urb *urb = NULL;
    char *buf = NULL;
    size_t writesize = min(count, (size_t)MAX_TRANSFER);
    dev = (struct usb_skel *)file->private_data;

    /* verify that we actually have some data to write */
    if (count == 0)
        goto exit;
    /* limit the number of URBs in flight to stop a user from using up all RAM */
    if (down_interruptible(&dev->limit_sem)) {
        retval = -ERESTARTSYS;
        goto exit;
    }
    spin_lock_irq(&dev->err_lock);
    if ((retval = dev->errors) < 0) {
        /* any error is reported once */
        dev->errors = 0;
```

```
            /* to preserve notifications about reset */
            retval = (retval == -EPIPE) ? retval : -EIO;
    }
    spin_unlock_irq(&dev->err_lock);
    if (retval < 0)
            goto error;
    /*当驱动有数据发送到 USB 设备时，首先分配一个 urb */
    urb = usb_alloc_urb(0, GFP_KERNEL);
    if (!urb) {
            retval = -ENOMEM;
            goto error;
    }
    /*以最有效的方式创建一个 DMA 缓冲区来发送数据到设备上，并复制数据至缓冲区*/
    buf = usb_buffer_alloc(dev->udev, writesize, GFP_KERNEL, &urb->transfer_dma);
    if (!buf) {
            retval = -ENOMEM;
            goto error;
    }
    if (copy_from_user(buf, user_buffer, writesize)) {
            retval = -EFAULT;
            goto error;
    }
    mutex_lock(&dev->io_mutex);
    if (!dev->interface) {              /* disconnect() was called */
            mutex_unlock(&dev->io_mutex);
            retval = -ENODEV;
            goto error;
    }
    /*在将 urb 提交给 USB 核心之前，正确初始化 urb */
    usb_fill_bulk_urb(urb, dev->udev,
                    usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
                    buf, writesize, skel_write_bulk_callback, dev);
    urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
    usb_anchor_urb(urb, &dev->submitted);
    /*提交 urb 给 USB 核心，由它将 urb 传递给设备*/
    retval = usb_submit_urb(urb, GFP_KERNEL);
    mutex_unlock(&dev->io_mutex);
    if (retval) {
            err("%s - failed submitting write urb, error %d", __func__, retval);
            goto error_unanchor;
    }

    /* release our reference to this urb, the USB core will eventually free it entirely */
    usb_free_urb(urb);

    return writesize;

error_unanchor:
    usb_unanchor_urb(urb);
error:
    if (urb) {
```

```
            usb_buffer_free(dev->udev, writesize, buf, urb->transfer_dma);
            usb_free_urb(urb);
    }
    up(&dev->limit_sem);

exit:
    return retval;
}
//当 urb 被成功传递到 USB 设备（或者在传输中发生了错误）时，则 urb 回调函数将被 USB 核心调用，也就是上面初始化 urb 中的 skel_write_bulk_callback
static void skel_write_bulk_callback(struct urb *urb)
{
    struct usb_skel *dev;
    dev = urb->context;
    /*检查 urb 的状态，判断这个 urb 是否成功完成传输*/
    if (urb->status) {
        if(!(urb->status == -ENOENT ||
          urb->status == -ECONNRESET ||
          urb->status == -ESHUTDOWN))
            err("%s - nonzero write bulk status received: %d",
              __func__, urb->status);

        spin_lock(&dev->err_lock);
        dev->errors = urb->status;
        spin_unlock(&dev->err_lock);
    }
    /*释放分配给这个 urb 的缓冲区*/
    usb_buffer_free(urb->dev, urb->transfer_buffer_length,
            urb->transfer_buffer, urb->transfer_dma);
    up(&dev->limit_sem);
}
```

由此可见，函数 skel_write_bulk_callback()的主要功能是对 urb->status 进行判断，根据错误提示显示出详细的错误信息，然后释放 urb 空间。

13.3.8 获取 USB 的接口

函数 skel_open()的功能是根据 usb_driver 和次设备号通过函数 usb_find_interface()获取 USB 的接口，然后通过函数 usb_get_intfdata()获取接口的私有数据，并将该数据赋值给 file->private_data = dev。函数 skel_open()的具体实现代码如下所示。

```
static int skel_open(struct inode *inode, struct file *file)
{
        struct usb_skel *dev;
        struct usb_interface *interface;
        int subminor;
        int retval = 0;

        subminor = iminor(inode);

        interface = usb_find_interface(&skel_driver, subminor);
```

```
	if (!interface) {
		pr_err("%s - error, can't find device for minor %d\n",
			__func__, subminor);
		retval = -ENODEV;
		goto exit;
	}

	dev = usb_get_intfdata(interface);
	if (!dev) {
		retval = -ENODEV;
		goto exit;
	}

	retval = usb_autopm_get_interface(interface);
	if (retval)
		goto exit;

	/* increment our usage count for the device */
	kref_get(&dev->kref);

	/* save our object in the file's private structure */
	file->private_data = dev;

exit:
	return retval;
}
```

13.3.9　释放不需要的资源

函数 skel_release()的功能是释放任何不需要的资源，减少在函数 skel_open()中增加的引用计数。函数 skel_release()的具体实现代码如下所示。

```
static int skel_release(struct inode *inode, struct file *file)
{
	struct usb_skel *dev;

	dev = file->private_data;
	if (dev == NULL)
		return -ENODEV;

	/* allow the device to be autosuspended */
	mutex_lock(&dev->io_mutex);
	if (dev->interface)
		usb_autopm_put_interface(dev->interface);
	mutex_unlock(&dev->io_mutex);

	/* decrement the count on our device */
	kref_put(&dev->kref, skel_delete);
	return 0;
}
```

13.3.10 字符设备函数

USB 骨架程序的字符设备函数 skel_read()的具体实现代码如下所示。

```
static ssize_t skel_read(struct file *file, char *buffer, size_t count,
                          loff_t *ppos)
{
        struct usb_skel *dev;
        int rv;
        bool ongoing_io;
        dev = (struct usb_skel *)file->private_data;            //获得文件私有数据
        if (!dev->bulk_in_urb || !count)                        //正在写的时候禁止读操作
                return 0;
        rv = mutex_lock_interruptible(&dev->io_mutex);          //获得锁
        if (rv < 0)
                return rv;
        if (!dev->interface) {
                rv = -ENODEV;
                goto exit;
        }
retry:
        spin_lock_irq(&dev->err_lock);
        ongoing_io = dev->ongoing_read;
        spin_unlock_irq(&dev->err_lock);
        if (ongoing_io) {                                       //USB 核正在读取数据中，数据没准备好
                if (file->f_flags & O_NONBLOCK) {               //如果为非阻塞，则结束
                        rv = -EAGAIN;
                        goto exit;
                }
                rv = wait_for_completion_interruptible(&dev->bulk_in_completion);   //等待
                if (rv < 0)
                        goto exit;
                dev->bulk_in_copied = 0;                        //复制到用户空间操作已成功
                dev->processed_urb = 1;                         //目前已处理好 urb
        }
        if (!dev->processed_urb) {                              //目前还没处理好 urb
                wait_for_completion(&dev->bulk_in_completion);  //等待完成
                dev->bulk_in_copied = 0;                        //复制到用户空间操作已成功
                dev->processed_urb = 1;                         //目前已处理好 urb
        }
        rv = dev->errors;
        if (rv < 0) {
                dev->errors = 0;
                rv = (rv == -EPIPE) ? rv : -EIO;
                dev->bulk_in_filled = 0;
                goto exit;
        }
        if (dev->bulk_in_filled) {                              //缓冲区有内容
        //可读数据大小为缓冲区内容减去已经复制到用户空间的数据大小
                size_t available = dev->bulk_in_filled - dev->bulk_in_copied;
```

```
                size_t chunk = min(available, count);          //真正读取数据大小
                if (!available) {
                        rv = skel_do_read_io(dev, count);      //没可读数据则调用 I/O 操作
                        if (rv < 0)
                                goto exit;
                        else
                                goto retry;
                }
                //复制缓冲区数据到用户空间
                if (copy_to_user(buffer, dev->bulk_in_buffer + dev->bulk_in_copied,chunk))   rv = -EFAULT;
                else
                        rv = chunk;
                dev->bulk_in_copied += chunk;                  //目前复制完成的数据大小
                if (available < count)                         //剩下可用数据小于用户需要的数据
                        skel_do_read_io(dev, count - chunk);   //调用 I/O 操作
        } else {
                rv = skel_do_read_io(dev, count);              //缓冲区无数据则调用 I/O 操作
                if (rv < 0)
                        goto exit;
                else if (!file->f_flags & O_NONBLOCK)
                        goto retry;
                rv = -EAGAIN;
        }
exit:
        mutex_unlock(&dev->io_mutex);
        return rv;
}
```

13.3.11　读取的数据量

通过函数 skel_read()的实现代码可知，在读取数据时如果发现在缓冲区中没有数据，或者缓冲区的数据小于用户需要读取的数据量时，则会调用 I/O 操作函数 skel_do_read_io()。函数 skel_do_read_io()的具体实现代码如下所示。

```
static int skel_do_read_io(struct usb_skel *dev, size_t count)
{
        int rv;
        usb_fill_bulk_urb(dev->bulk_in_urb,dev->udev,usb_rcvbulkpipe(dev->udev,
                                dev->bulk_in_endpointAddr),dev->bulk_in_buffer,
                        min(dev->bulk_in_size, count),skel_read_bulk_callback,dev); //填充 urb
        spin_lock_irq(&dev->err_lock);
        dev->ongoing_read = 1;     //标志正在读取数据中
        spin_unlock_irq(&dev->err_lock);
        rv = usb_submit_urb(dev->bulk_in_urb, GFP_KERNEL);     //提交 urb
        if (rv < 0) {
                err("%s - failed submitting read urb, error %d",
                        __func__, rv);
                dev->bulk_in_filled = 0;
                rv = (rv == -ENOMEM) ? rv : -EIO;
                spin_lock_irq(&dev->err_lock);
                dev->ongoing_read = 0;
```

```
                spin_unlock_irq(&dev->err_lock);
        }
        return rv;
}
```

通过函数 skel_do_read_io()的实现代码可知，它只是完成了 urb 的填充和提交工作。当 USB 核心读取到数据以后，会调用填充 urb 时设置的回调函数 skel_read_bulk_callback()。函数 skel_read_bulk_ callback()的具体实现代码如下所示。

```
static void skel_read_bulk_callback(struct urb *urb)
{
        struct usb_skel *dev;
        dev = urb->context;
        spin_lock(&dev->err_lock);
        if (urb->status) {                                      //根据返回状态判断是否出错
                if (!(urb->status == -ENOENT ||
                    urb->status == -ECONNRESET ||
                    urb->status == -ESHUTDOWN))
                        err("%s - nonzero write bulk status received: %d",
                            __func__, urb->status);
                dev->errors = urb->status;
        } else {
                dev->bulk_in_filled = urb->actual_length;       //记录缓冲区的大小
        }
        dev->ongoing_read = 0;                                  //已经读取数据完毕
        spin_unlock(&dev->err_lock);
        complete(&dev->bulk_in_completion);                     //唤醒 skel_read()函数
}
```

到此为止，USB 驱动框架文件 usb-skeleton.c 全部分析完毕。在此简单总结下具体实现流程。

（1）首先在模块加载中注册 usb_driver。

（2）然后在 probe()函数中初始化一些参数，最重要的是注册了 USB 设备，这个 USB 设备相当于一个字符设备，提供 file_operations 接口。

（3）然后分别设计函数 open、close、read、write，此处的 open()函数基本没做什么事情，在函数 write()中通过分配 urb、填充 urb 和提交 urb，实现如下两种功能。

☑ 读取 urb 分配在 probe 中的申请空间。

☑ 写入 urb 分配在 write 中的申请空间。

13.4 实 战 演 练

通过本章前面内容的学习，已经了解了 USB Gadget 驱动基本架构知识。本节将通过具体实例剖析移植 USB Gadget 驱动的具体方法。

13.4.1 移植 USB Gadget 驱动

下面将以 Linux 2.6.37.4 内核为基础，介绍在 XC2440 开发板上移植 USB Gadget 驱动的方法。

（1）因为需要在文件 mach-xc2440.c 中添加对 USB Gadget 驱动的支持，所以加入如下头文件。

```
#include <plat/udc.h>
```



在结构体 xc2440_devices 中加入下面的值。

```
&s3c_device_usbgadget,
```

然后构建 USB Gadget 设备平台的数据结构，具体代码如下所示。

```
static void xc2440_udc_pullup(enum s3c2410_udc_cmd_e cmd)
    {
        switch (cmd) {
            case S3C2410_UDC_P_ENABLE :
                gpio_set_value(S3C2410_GPG(12), 1);
                break;
            case S3C2410_UDC_P_DISABLE :
                gpio_set_value(S3C2410_GPG(12), 0);
                break;
            case S3C2410_UDC_P_RESET :
                break;
            default:
                break;
        }
    }

static struct s3c2410_udc_mach_info xc2440_udc_cfg __initdata = {
.udc_command = xc2440_udc_pullup,
};
```

（2）在函数 xc2440_machine_init()中加入如下代码。

```
s3c24xx_udc_set_platdata(&xc2440_udc_cfg);
```

然后配置内核以支持 USB Gadget 驱动。

```
Device Drivers --->
     [*] USB support --->
    <*> USB Gadget Support --->
    USB Peripheral Controller (S3C2410 USB Device Controller) --->
S3C2410 USB Device Controller
<M> USB Gadget Drivers
<M> File-backed Storage Gadget
[*] File-backed Storage Gadget testing version
<M> Mass Storage Gadget
```

（3）启动内核后会输出：

```
s3c2440-usbgadget s3c2440-usbgadget: S3C2440: increasing FIFO to 128 bytes
```

（4）开始进行编译工作，编译命令如下所示。

```
#make M=drivers/usb/gadget modules
```

编译后会在 drivers/usb/gadget 目录下生成 g_file_storage.ko 和 g_mass_storage.ko 两个模块文件。

将上述两个文件下载到开发板的文件系统中，将文件保存到/lib/modules/2.6.37.4 目录下。

（5）执行如下命令：

```
#insmod /lib/modules/2.6.37.4/g_file_storage.ko file=/dev/sda1
```

这样就完成了整个驱动程序的移植工作，输出信息如下所示。

```
[root@XC2440 /]# insmod /lib/modules/2.6.37.4/g_file_storage.ko file=/dev/sda1
g_file_storage gadget: No serial-number string provided!
g_file_storage gadget: File-backed Storage Gadget, version: 1 September 2010
g_file_storage gadget: Number of LUNs=1
g_file_storage gadget-lun0: ro=0, nofua=0, file: /dev/sda1
g_file_storage gadget: full speed config #1
g_file_storage gadget: full speed config #1
```

13.4.2 移植 USB HOST 驱动

下面将详细讲解在 XC2440 开发板上移植 USB HOST 驱动的方法，具体实现流程如下。

（1）为了在文件 mach-xc2440.c 中添加 USB Host 驱动的支持，需要在结构体 xc2440_devices 中加入下面的值。

```
&s3c_device_ohci,
```

（2）在文件/arch/arm/plat-samsung/dev-usb.c 中定义 s3c_device_ohci 结构体，因为系统默认没有对它进行编译支持，所以修改同目录下的 Kconfig 文件。

```
config S3C_DEV_USB_HOST
    bool
    default y
    help
        Compile in platform device definition for USB host.
```

或者修改文件 arch/arm/mach-s3c2440/Kconfig。

```
config MACH_XC2440
    bool "XC2440 development board with S3C2440 CPU module"
    select CPU_S3C2440
    select S3C_DEV_NAND
    select S3C_DEV_USB_HOST
    help
        Say Y here if you are using the XC2440 development board.
```

（3）开始配置内核，目的是支持 USB Host 驱动。

```
Device drivers --->
    SCSI Device support --->
        <*> SCSI device support
<*> SCSI disk support
     [*] HID Devices --->
-*- Generic HID support
<*> USB Human Interface Device (full HID) support
     [*] USB support --->
{*} Support for Host-side USB
[*] USB announce new devices
[*] USB device filesystem
<*> OHCI HCD support
<*> USB Mass Storage support
```

其中 USB Human Interface Device (full HID) support 是对 USB 鼠标键盘的支持，而 SCSI disk support 和 USB Mass Storage support 表示对 U 盘的支持。

（4）开始移植测试工作，系统启动时会输出如下调试信息。

```
s3c2410-ohci s3c2410-ohci: S3C24XX OHCI
s3c2410-ohci s3c2410-ohci: new USB bus registered, assigned bus number 1
s3c2410-ohci s3c2410-ohci: irq 42, io mem 0x49000000
usb usb1: New USB device found, idVendor=1d6b, idProduct=0001
usb usb1: New USB device strings: Mfr=3, Product=2, SerialNumber=1
usb usb1: Product: S3C24XX OHCI
usb usb1: Manufacturer: Linux 2.6.37.4 ohci_hcd
```

第 14 章　Time Device 驱动

Time Device 是 Android 系统中的一个定时设备驱动，对 Android 移动设备提供了定时控制的功能。Time Device 分为 Timed Output 和 Timed Gpio 两类。本章将详细讲解 Android 系统中定时设备驱动 Time Device 模块的基本知识，具体分析其原理和实现源码，为读者学习本书后面的知识打下基础。

14.1　Timed Output 驱动架构

在 Android 系统中，Timed Output 驱动是一个很重要的框架，例如经常通过 Timed Output 驱动程序框架来实现 Vibrator（振动）驱动程序。Timed Output 驱动是基于 sys 文件系统实现的，能够对设备进行定时控制功能，目前支持设备有 Vibrator（振动）和 LED（闪光灯）设备。Timed Output 驱动会注册 sys/class/timed_output/目录，每一个注册实现的 Timed Output 设备（例如 Vibrator 和 LED）将会在 sys/class/timed_output/目录中新建一个和设备同名的子目录。在子目录中有一个名为 enable 的子文件，通过对此文件的读写实现对设备的控制和显示功能。

在 Android 系统中，有如下两个实现 Timed Output 驱动的文件。

☑　drivers/staging/android/timed_output.c

☑　drivers/staging/android/timed_output.h

本节将详细分析上述文件的具体实现过程。

14.1.1　设备类

在 Linux 内核中定义了一个名为 struct class 的结构体，表示一个 struct class 结构体类型变量对应一个类，并且在内核中还提供了函数 class_create(…)，功能是创建一个放于 sysfs 下面的类。在创建好这个类之后，调用函数 device_create(…)在/dev 目录下创建相应的设备节点。当加载模块时，保存在用户空间中的 udev 会自动响应函数 device_create(…)，并在/sysfs 目录下寻找对应的类以创建设备节点。

在 Linux 系统中，和设备类相关的 API 的具体说明如下。

☑　class_destroy(class)：功能是将 class 注册到内核中，在调用此类之前必须手工分配 class 内存，在调用之后必须设置 class 的 name 等参数。

☑　class_create(owner,name)：功能是创建 class，并将 class 注册到内核中，返回 class 结构体指针。

☑　void class_unregister(struct class *cls)：功能是注销 class，此 API 函数经常与 class_register 配对使用。

☑　void class_destory(struct class *cls)：功能是注销 class，此 API 函数经常与 class_create 配对使用。

例如在下面的代码中，演示了通过 class_create()、class_destroy()注册并注销/sys/class/my_char_dev 的过程。

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/device.h>

struct class *mem_class;
```

```
static int __init class_create_destroy_init(void)
{
	// class_create 动态创建设备的逻辑类并完成部分字段的初始化，然后将其添加到内核中。创建的逻辑类位于/sys/class/下
	// 参数：
	//		owner，拥有者。一般赋值为 THIS_MODULE
	//		name，创建的逻辑类名称
	mem_class = class_create(THIS_MODULE, "my_char_dev");
	if (mem_class==NULL)
	{
		printk("<0> create class failed!\n");
		return -1;
	}

	return 0;
}

static void __exit class_create_destroy_exit(void)
{
	if (mem_class != NULL)
	{
		class_destroy(mem_class);
		mem_class = NULL;
	}

}

module_init(class_create_destroy_init);
module_exit(class_create_destroy_exit);

MODULE_LICENSE("GPL");
```

例如在下面的代码中，演示了通过函数 class_register()和 class_unregister()注册并注销/sys/class/my_char_dev 的过程。

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/device.h>
#include <linux/slab.h>

#define CLASS_NAME "my_char_dev"
struct class *mem_class;

static void class_create_release (struct class *cls)
{
	printk("%s\n", __func__ );
	kfree(cls);
}

static int __init class_create_destroy_init(void)
{
	printk("%s\n", __func__);
```

```
    int ret;

    //申请 class 结构体内存
    mem_class = kzalloc(sizeof(*mem_class), GFP_KERNEL);
    if (mem_class == NULL)
    {
        printk("create mem class failed!\n");
        return -1;
    }
    printk("create mem class success\n");

    mem_class->name = CLASS_NAME;
    mem_class->owner = THIS_MODULE;
    //注销 class 时的回调函数，在此回调函数中释放之前所分配的 class 结构体内存
    mem_class->class_release = class_create_release;

    //将 class 注册到内核中，同时会在/sys/class/下创建 class 对应的节点
    int retval = class_register(mem_class);
    if (ret)
    {
        printk("class_register failed!\n");
        kfree(mem_class);
        return -1;
    }
    printk("class_register success\n");


    return 0;
}

static void __exit class_create_destroy_exit(void)
{
    printk("%s\n", __func__);

    if (mem_class != NULL)
    {
        class_unregister(mem_class);
        mem_class = NULL;
    }
}

module_init(class_create_destroy_init);
module_exit(class_create_destroy_exit);

MODULE_LICENSE("GPL");
```

通过查看 class_create()和 class_register()、class_destroy()和 class_unregister()的具体源码可知，上述两段演示代码的功能是等价的，其中 class_register()的具体实现代码如下所示。

```
//将 class 注册到/sys/class/中
#define class_register(class)                \
({                                 \
```

```
        static struct lock_class_key __key; \
        __class_register(class, &__key);    \
})
```

函数 class_register()是通过调用函数__class_register()实现注册到 sysfs 中这一功能的，函数__class_register()的具体实现代码如下所示。

```
int __class_register(struct class *cls, struct lock_class_key *key)
{
        struct class_private *cp;
        int error;

        pr_debug("device class '%s': registering\n", cls->name);

        cp = kzalloc(sizeof(*cp), GFP_KERNEL);
        if (!cp)
                return -ENOMEM;
        klist_init(&cp->class_devices, klist_class_dev_get, klist_class_dev_put);
        INIT_LIST_HEAD(&cp->class_interfaces);
        kset_init(&cp->class_dirs);
        __mutex_init(&cp->class_mutex, "struct class mutex", key);
        error = kobject_set_name(&cp->class_subsys.kobj, "%s", cls->name);
        if (error) {
                kfree(cp);
                return error;
        }

        /* set the default /sys/dev directory for devices of this class */
        if (!cls->dev_kobj)
                cls->dev_kobj = sysfs_dev_char_kobj;

#if defined(CONFIG_SYSFS_DEPRECATED) && defined(CONFIG_BLOCK)
        /* let the block class directory show up in the root of sysfs */
        if (cls != &block_class)
                cp->class_subsys.kobj.kset = class_kset;
#else
        cp->class_subsys.kobj.kset = class_kset;
#endif
        cp->class_subsys.kobj.ktype = &class_ktype;
        cp->class = cls;
        cls->p = cp;

        //将 class 注册到内核中
        error = kset_register(&cp->class_subsys);
        if (error) {
                kfree(cp);
                return error;
        }
        error = add_class_attrs(class_get(cls));
        class_put(cls);
        return error;
}
```

函数 class_unregister()的具体实现代码如下所示。

```
void class_unregister(struct class *cls)
{
	pr_debug("device class '%s': unregistering\n", cls->name);
	remove_class_attrs(cls);
	//将 class 从内核中注销
	kset_unregister(&cls->p->class_subsys);
}
```

函数 class_create()的具体实现代码如下所示。

```
#define class_create(owner, name)		\
({						\
	static struct lock_class_key __key; \
	__class_create(owner, name, &__key);	\
})
```

在 Android 系统中，函数 class_create()通过调用__class_create()函数注册到内核中。函数__class_create()的具体实现代码如下所示。

```
struct class *__class_create(struct module *owner, const char *name,
			     struct lock_class_key *key)
{
	struct class *cls;
	int retval;

	//分配 class 结构体
	cls = kzalloc(sizeof(*cls), GFP_KERNEL);
	if (!cls) {
		retval = -ENOMEM;
		goto error;
	}

	cls->name = name;
	cls->owner = owner;
	//class 对应的释放函数，在 class 从内核中注销时会执行该函数
	cls->class_release = class_create_release;

	//通过调用__class_register()将 class 注册到内核中
	retval = __class_register(cls, key);
	if (retval)
		goto error;

	return cls;

error:
	kfree(cls);
	return ERR_PTR(retval);
}
```

函数 class_create_release()的具体实现代码如下所示。

```
static void class_create_release(struct class *cls)
 {
	pr_debug("%s called for %s\n", __func__, cls->name);
	//释放 class 结构体
```

```
        kfree(cls);
}
```

因为在 Android 系统中，__class_create()是通过调用__class_register()注册到 sysfs 中的，所以函数 class_create()和 class_register()的作用是相似的。函数 class_destroy()的具体实现代码如下所示。

```
void class_destroy(struct class *cls)
{
    if ((cls == NULL) || (IS_ERR(cls)))
        return;
    //调用 class_unregister()将 class 从内核中注销
    class_unregister(cls);
}
```

在 Android 系统中，函数 class_destroy()是通过调用 class_unregister()实现的。

14.1.2 分析 Timed Output 驱动的具体实现

1. 文件 timed_output.h

在文件 timed_output.h 中定义了结构体 timed_output_dev，使设备设置定时器功能，并设置返回定时器的剩余时间。文件 timed_output.h 的实现代码如下所示。

```
struct timed_output_dev {
    const char*name;

    /*设置定时器功能*/
    void  (*enable)(struct timed_output_dev *sdev, int timeout);

    /*返回在定时器的毫秒的当前数量*/
    int         (*get_time)(struct timed_output_dev *sdev);

    /* 私有数据 */
    struct device    *dev;
    int         index;
    int         state;
};
extern int timed_output_dev_register(struct timed_output_dev *dev);
extern void timed_output_dev_unregister(struct timed_output_dev *dev);
```

2. 分析文件 timed_output.c

timed_output 属于 Android 系统中的一个内核模块，文件 timed_output.c 是文件 timed_output.h 的具体实现，其实现代码如下所示。

（1）在文件 timed_output.c 中，注册和注销 Timed Output 驱动的实现代码如下所示。

```
static int __init timed_output_init(void)
{
        return create_timed_output_class();
}

static void __exit timed_output_exit(void)
{
        class_destroy(timed_output_class);
```

```
}

module_init(timed_output_init);
module_exit(timed_output_exit);
```

在上述代码中，module_init(timed_output_init)和 module_exit(timed_output_exit)是两个宏，这两个宏左右是注册内核模块的开始和结束函数。通过上述实现代码可知，是通过这两个宏向一个数据段中添加和删除这两个函数指针来具体实现的。

（2）在初始化函数 timed_output_init 中调用了函数 create_timed_output_class()，在此函数中用到了原子操作代码“atomic_set(&device_count, 0);”，功能是为 Timed Output 设备创建了一个功能类，具体实现代码如下所示。

```
static int create_timed_output_class(void)
{
	if (!timed_output_class) {
		timed_output_class = class_create(THIS_MODULE, "timed_output");
		if (IS_ERR(timed_output_class))
			return PTR_ERR(timed_output_class);
		atomic_set(&device_count, 0);
		timed_output_class->dev_groups = timed_output_groups;
	}

	return 0;
}
```

而在函数__exit timed_output_exit()中，通过调用 class_destroy()函数销毁了/sys/class 下的类。当在 Linux 系统中动态创建了设备节点之后，在卸载时需要使用 class_destroy()函数注销。

注意：原子操作

原子操作是指该操作不会在执行完毕前被任何其他任务或事件打断，也就是说，它是最小的执行单位，不可能有比它更小的执行单位，因此这里的原子实际是使用了物理学里的物质微粒的概念。原子操作需要硬件的支持，因此是架构相关的，其和原子类型的定义都定义在内核源码树文件include/asm/atomic.h中，它们都使用汇编语言实现，因为语言并不能实现这样的操作。原子操作主要用于实现资源计数，在现实应用中的很多引用计数（refcnt）就是通过原子操作实现的。

（3）函数 timed_output_dev_register()的功能是注册 Timed Output 设备，具体实现代码如下所示。

```
int timed_output_dev_register(struct timed_output_dev *tdev)
{
	int ret;

	if (!tdev || !tdev->name || !tdev->enable || !tdev->get_time)
		return -EINVAL;

	ret = create_timed_output_class();
	if (ret < 0)
		return ret;

	tdev->index = atomic_inc_return(&device_count);
	tdev->dev = device_create(timed_output_class, NULL,
		MKDEV(0, tdev->index), NULL, "%s", tdev->name);
	if (IS_ERR(tdev->dev))
```

```
                return PTR_ERR(tdev->dev);

        dev_set_drvdata(tdev->dev, tdev);
        tdev->state = 0;
        return 0;
}
```

（4）函数 timed_output_dev_unregister()的功能是注销 Timed Output 设备，具体实现代码如下所示。

```
void timed_output_dev_unregister(struct timed_output_dev *tdev)
{
     device_remove_file(tdev->dev, &dev_attr_enable);
     device_destroy(timed_output_class, MKDEV(0, tdev->index));
     dev_set_drvdata(tdev->dev, NULL);
}
```

（5）函数 enable_show()和 enable_store()的功能是通过调用 sys 文件系统来实现驱动功能，在每个 Timed Output 设备中都有 enable 文件。当在写这个文件时，表示正在设置定时器时间并启动定时器。函数 enable_show()和 enable_store()的具体实现代码如下所示。

```
static ssize_t enable_show(struct device *dev, struct device_attribute *attr,
          char *buf)
{
     struct timed_output_dev *tdev = dev_get_drvdata(dev);
     int remaining = tdev->get_time(tdev);
     return sprintf(buf, "%d\n", remaining);
}
static ssize_t enable_store(
          struct device *dev, struct device_attribute *attr,
          const char *buf, size_t size)
{
     struct timed_output_dev *tdev = dev_get_drvdata(dev);
     int value;
     if (sscanf(buf, "%d", &value) != 1)
          return -EINVAL;
     tdev->enable(tdev, value);
     return size;
}
```

14.1.3　实战演练——实现设备的读写操作

在文件 kernel/drivers/staging/android/timed_output.c 中，timed_output_dev 是时间输出类的一个常用接口，其最大的特点是使用 timed_output_dev_register 进行注册，这样就会生成一个名为/sys/class/timed_output 的目录。接下来以一个振动器为例，讲解实现 timed_output_dev 设备的读写操作的流程。

（1）定义 timed_output_dev 类型设备的结构体 mt6573_vibrator，具体实现代码如下所示。

```
static struct timed_output_dev mt6573_vibrator =
{
     .name = "vibrator",
     .get_time = vibrator_get_time,
     .enable = vibrator_enable,
};
```

（2）在振动器初始化代码中完成注册功能，具体代码如下所示。

```
timed_output_dev_register(&mt6573_vibrator);
```

在/sys/class/timed_output/vibrator/目录中，按下 pwd 后即可在 ADB 中看到如图 14-1 所示的属性。

```
/sys/devices/virtual/timed_output/vibrator
# ls
ls
uevent
subsystem
power
enable
```

图 14-1　/sys/class/timed_output/vibrator/目录中的属性

（3）使用 echo 0 或者 1>enable 指令即可对振动器实现开关控制。

到此为止已完成了驱动层的开发工作，我们可以在上层设置一个测试程序进行验证。在上层中可以通过 write()函数调用 timed_output_dev 的函数接口 store，从而达到调用 enable 的目的。同样的道理，可以通过 read 得到属性 show 的值。具体测试代码如下所示。

```
#define THE_DEVICE "/sys/class/timed_output/vibrator/enable"
static int sendit(int timeout_ms)
{
	int nwr, ret, fd;
	char value[20];
	fd = open(THE_DEVICE, O_RDWR);
	if(fd < 0)
		return errno;
	nwr = sprintf(value, "%d\n", timeout_ms);
	ret = write(fd, value, nwr);
	close(fd);
	return (ret == nwr) ? 0 : -1;
}
int vibrator_on(int timeout_ms)
{
	return sendit(timeout_ms);
}
int vibrator_off()
{
	return sendit(0);
}
```

14.2　Timed Gpio 驱动架构

在 Android 系统中，Timed Gpio 是基于 Timed Output 模块的一个驱动程序，功能是定时控制 GPIO 以实现振动器的效果。Timed Gpio 可以调用 Timed Output 框架注册一个驱动程序。与传统的 GPIO 驱动相比，Timed Gpio 的最大特点是将普通的 GPIO 与内核定时器绑定在一起，实现了一种时钟控制的 GPIO。当定时器过期以后，GPIO 的状态会被设置为一个指定的状态。在 Android 系统中，Timed Gpio 的本质功能是通过 sysfs 操作 GPIO，例如可以让 GPIO 输出“高/低”电平，并同时指定一个定时器的过期时间。当到达过期时间后可以执行 callback（回调）函数，这样可以重新设置 GPIO 的输出电平。在 Android 系统中，Timed Gpio 驱动程序在如下两个文件中实现。

☑　drivers/staging/android/timed_gpio.h

☑　drivers/staging/android/timed_gpio.c

下面将详细讲解上述文件的具体实现流程。

14.2.1 分析文件 timed_gpio.h

文件 timed_gpio.h 比较简单，在里面定义了 Timed Gpio 驱动的名称，并设置结构体 timed_gpio 作为驱动的私有结构体。文件 timed_gpio.h 的实现代码如下所示。

```
#ifndef _LINUX_TIMED_GPIO_H
#define _LINUX_TIMED_GPIO_H
#define TIMED_GPIO_NAME "timed-gpio"//定义 Timed Gpio 驱动的名称
struct timed_gpio {
	const char *name;
	unsigned	gpio;
	int		max_timeout;
	u8		active_low;
};
struct timed_gpio_platform_data {
	int		num_gpios;
	struct timed_gpio *gpios;
};
#endif
```

14.2.2 分析文件 timed_gpio.c

文件 timed_gpio.c 实现了一个 Timed GPIO Driver，此功能是基于本章前面讲解的 timed_output.c 文件提供的功能实现的，最终目的是实现一个基于 Platform Driver 架构的驱动，并且提供标准的接口。文件 timed_gpio.c 的功能比较复杂，下面将讲解整个文件的具体实现流程。

（1）首先看结构体 timed_gpio_data，功能是把一个 GPIO 设备与一个 hrtimer 定时器相互关联。具体实现代码如下所示。

```
struct timed_gpio_data {
		struct timed_output_dev dev;
		struct hrtimer timer;
		spinlock_t lock;
		unsigned gpio;
		int max_timeout;
		u8 active_low;
};
```

在上述代码中，最后一个成员变量 active_low 有多个含义，其中在 probe 阶段会根据这个变量设置 GPIO 输出的电平，此时该变量类似于指定 GPIO 在初始化时的默认电平。但是如果在后续的使用过程中，当通过 sysfs 的 enable()函数设置 GPIO 输出电平时，会将这个变量作为一个标志来使用。如果 active_low!=0，则反转输出电平极性，否则不反转。

假如正在调用函数 gpio_enable(struct timed_output_dev *dev,int value)，此时传进来的参数 value 的值是 1，那么如果 active_low==0，则 GPIO 引脚输出高电平；反之如果 active_low 不等于 0，则输出的是低电平。

（2）使用文件 timed_output.c 初始化函数实现初始化操作，使用注销函数实现注销操作。也就是说，通过如下两个函数分别实现对驱动设备的注册和注销。

```
int timed_output_dev_register(struct timed_output_dev *tdev)
{
```

```
        int ret;
        if (!tdev || !tdev->name || !tdev->enable || !tdev->get_time)
                return -EINVAL;
        ret = create_timed_output_class();
        if (ret < 0)
                return ret;
        tdev->index = atomic_inc_return(&device_count);
        tdev->dev = device_create(timed_output_class, NULL,
                MKDEV(0, tdev->index), NULL, tdev->name);
        if (IS_ERR(tdev->dev))
                return PTR_ERR(tdev->dev);
        ret = device_create_file(tdev->dev, &dev_attr_enable);
        if (ret < 0)
                goto err_create_file;
        dev_set_drvdata(tdev->dev, tdev);
        tdev->state = 0;
        return 0;
err_create_file:
        device_destroy(timed_output_class, MKDEV(0, tdev->index));
        printk(KERN_ERR "timed_output: Failed to register driver %s\n",
                        tdev->name);
        return ret;
}
EXPORT_SYMBOL_GPL(timed_output_dev_register);
void timed_output_dev_unregister(struct timed_output_dev *tdev)
{
        device_remove_file(tdev->dev, &dev_attr_enable);
        device_destroy(timed_output_class, MKDEV(0, tdev->index));
        dev_set_drvdata(tdev->dev, NULL);
}
EXPORT_SYMBOL_GPL(timed_output_dev_unregister);
```

（3）再看 probe 函数 timed_gpio_probe()，其具体功能如下。

- ☑ 分配 num_gpios 个 timed_gpio_data 结构体，每个分别对应需要管理的一个 GPIO。
- ☑ 为每一个 GPIO 调用 hrtimer_init 初始化的内核定时器，并且设置定时器过期的 Callback Handler（回调函数）为 gpio_timer_func。
- ☑ 为每一个 GPIO 初始化结构体 timed_gpio_data 其他成员变量，设置 enable()函数为 gpio_enable，设置 get_time()函数为 gpio_get_time。
- ☑ 为每一个 GPIO 调用函数 timed_output_dev_register()（此函数由 timed_output.c 提供）创建 sysfs 设备文件，并创建 struct device 对象。
- ☑ 为每一个 GPIO 调用 gpio_direction_output 设置其初始输出电平。

函数 timed_gpio_probe()的具体实现代码如下所示。

```
static int timed_gpio_probe(struct platform_device *pdev)
{
        struct timed_gpio_platform_data *pdata = pdev->dev.platform_data;
        struct timed_gpio *cur_gpio;
        struct timed_gpio_data *gpio_data, *gpio_dat;
        int i, ret;

        if (!pdata)
```

```
                return -EBUSY;

        gpio_data = kzalloc(sizeof(struct timed_gpio_data) * pdata->num_gpios,
                        GFP_KERNEL);
        if (!gpio_data)
                return -ENOMEM;

        for (i = 0; i < pdata->num_gpios; i++) {
                cur_gpio = &pdata->gpios[i];
                gpio_dat = &gpio_data[i];

                hrtimer_init(&gpio_dat->timer, CLOCK_MONOTONIC,
                                HRTIMER_MODE_REL);
                gpio_dat->timer.function = gpio_timer_func;
                spin_lock_init(&gpio_dat->lock);

                gpio_dat->dev.name = cur_gpio->name;
                gpio_dat->dev.get_time = gpio_get_time;
                gpio_dat->dev.enable = gpio_enable;
                ret = gpio_request(cur_gpio->gpio, cur_gpio->name);
                if (ret < 0)
                        goto err_out;
                ret = timed_output_dev_register(&gpio_dat->dev);
                if (ret < 0) {
                        gpio_free(cur_gpio->gpio);
                        goto err_out;
                }

                gpio_dat->gpio = cur_gpio->gpio;
                gpio_dat->max_timeout = cur_gpio->max_timeout;
                gpio_dat->active_low = cur_gpio->active_low;
                gpio_direction_output(gpio_dat->gpio, gpio_dat->active_low);
        }

        platform_set_drvdata(pdev, gpio_data);

        return 0;

err_out:
        while (--i >= 0) {
                timed_output_dev_unregister(&gpio_data[i].dev);
                gpio_free(gpio_data[i].gpio);
        }
        kfree(gpio_data);

        return ret;
}
```

再看结构体 timed_gpio_probe，具体实现代码如下所示。

```
static struct platform_driver timed_gpio_driver = {
        .probe          = timed_gpio_probe,
        .remove         = timed_gpio_remove,
```

```
        .driver         = {
                .name           = TIMED_GPIO_NAME,
                .owner          = THIS_MODULE,
        },
};
```

而函数 timed_gpio_remove()实现的功能正好与函数 timed_gpio_driver()相反，具体实现代码如下所示。

```
static int timed_gpio_remove(struct platform_device *pdev)
{
        struct timed_gpio_platform_data *pdata = pdev->dev.platform_data;
        struct timed_gpio_data *gpio_data = platform_get_drvdata(pdev);
        int i;

        for (i = 0; i < pdata->num_gpios; i++) {
                timed_output_dev_unregister(&gpio_data[i].dev);
                gpio_free(gpio_data[i].gpio);
        }

        kfree(gpio_data);

        return 0;
}
```

（4）接下来看函数 gpio_timer_func()，这是一个定时器的 handler 函数，功能是如果设置的 hrtimer 的 timeout 时间到则会自动被内核调用，通过调用 gpio_direction_output()函数让 GPIO 引脚输出相应的电平值。函数 gpio_timer_func()的具体实现代码如下所示。

```
static enum hrtimer_restart gpio_timer_func(struct hrtimer *timer)
{
        struct timed_gpio_data *data =
                container_of(timer, struct timed_gpio_data, timer);

        gpio_direction_output(data->gpio, data->active_low ? 1 : 0);
        return HRTIMER_NORESTART;
}
```

通过上述代码可知，函数 gpio_timer_func()设置的 GPIO 输出电平值取决于 timed_gpio_data->active_low。如果 active_low!=0，则输出高电平，否则输出低电平。

（5）再看函数 gpio_enable()，功能是首先根据参数 value 输出 GPIO 的电平，然后用参数 value 重新设置并重新启动 hrtimer。这样当在 value 指定的 timeout 时间到达后，会再次触发调用 gpio_timer_func()函数。函数 gpio_enable()的具体实现代码如下所示。

```
static void gpio_enable(struct timed_output_dev *dev, int value)
{
        struct timed_gpio_data  *data =
                container_of(dev, struct timed_gpio_data, dev);
        unsigned long   flags;

        spin_lock_irqsave(&data->lock, flags);

        /* cancel previous timer and set GPIO according to value */
        hrtimer_cancel(&data->timer);
        gpio_direction_output(data->gpio, data->active_low ? !value : !!value);

```

```
	if (value > 0) {
		if (value > data->max_timeout)
			value = data->max_timeout;

		hrtimer_start(&data->timer,
			ktime_set(value / 1000, (value % 1000) * 1000000),
			HRTIMER_MODE_REL);
	}

	spin_unlock_irqrestore(&data->lock, flags);
}
```

通过上述实现代码可知，参数 value 有如下两个含义。

☑ 作为 GPIO 输出的电平，如果 value 不等于 0，则输出高电平，否则输出低电平。

☑ value 可以被用作重置 hrtimer 定时器的 timeout 时间值。

（6）再看函数 gpio_get_tim()，功能是调用函数 hrtimer_get_remaining()得到 Timed GPIO 关联 hrtimer 的剩余时间。函数 gpio_get_tim()的具体实现代码如下所示。

```
static int gpio_get_time(struct timed_output_dev *dev)
{
	struct timed_gpio_data	*data =
		container_of(dev, struct timed_gpio_data, dev);

	if (hrtimer_active(&data->timer)) {
		ktime_t r = hrtimer_get_remaining(&data->timer);
		struct timeval t = ktime_to_timeval(r);
		return t.tv_sec * 1000 + t.tv_usec / 1000;
	} else
		return 0;
}
```

而函数 hrtimer_get_remaining()在文件/kernel/hrtimer.c 中定义，具体实现代码如下所示。

```
ktime_t hrtimer_get_remaining(const struct hrtimer *timer)
{
	unsigned long flags;
	ktime_t rem;

	lock_hrtimer_base(timer, &flags);
	rem = hrtimer_expires_remaining(timer);
	unlock_hrtimer_base(timer, &flags);

	return rem;
}
```

到此为止，分析 Android 系统驱动 Timed Gpio 的工作全部结束。

注意： 用户接口Timed GPIO Driver基于标准Linux设备模型和Platform Driver框架，也是利用sysfs文件系统透露出两个标准接口show和store，对应的实现文件为sys/class/timed_output//enable，具体应用方面的工作可以通过sysfs与其交互的方式实现。

第 15 章　警报器系统驱动 Alarm

Alarm 是 Android 系统中的一个硬件时钟，也被称为警报器系统，功能是提供一个定时器把设备从睡眠状态唤醒，同时提供一个在设备睡眠时仍然会运行的时钟基准。本章将详细讲解 Android 系统中的警报器系统驱动 Alarm 的基本架构知识，为读者学习本书后面的知识打下基础。

15.1　Alarm 系统基础

在 Android 系统中，警报器系统又叫时钟系统或闹钟系统，Alarm 闹钟是 Android 系统中在标准 RTC 驱动上开发的一个新的驱动，提供了一个定时器用于把设备从睡眠状态唤醒，当然因为它是依赖 RTC 驱动的，所以它同时还可以为系统提供一个掉电下还能运行的实时时钟。当系统断电时，主板上的 RTC 芯片将继续维持系统的时间，这样保证再次开机后系统的时间不会错误。当系统开始时，内核从 RTC 中读取时间来初始化系统时间，关机时又将系统时间写回到 RTC 中，关机阶段将由主板上另外的电池来供应 RTC 计时。Android 中的 Alarm 在设备处于睡眠模式时仍保持活跃，它可以设置唤醒设备。本节将详细讲解 Android 系统中 Alarm 报警系统的基本知识。

15.1.1　Alarm 层次结构介绍

Android 平台中 Alarm 系统的基本层次结构如图 15-1 所示。

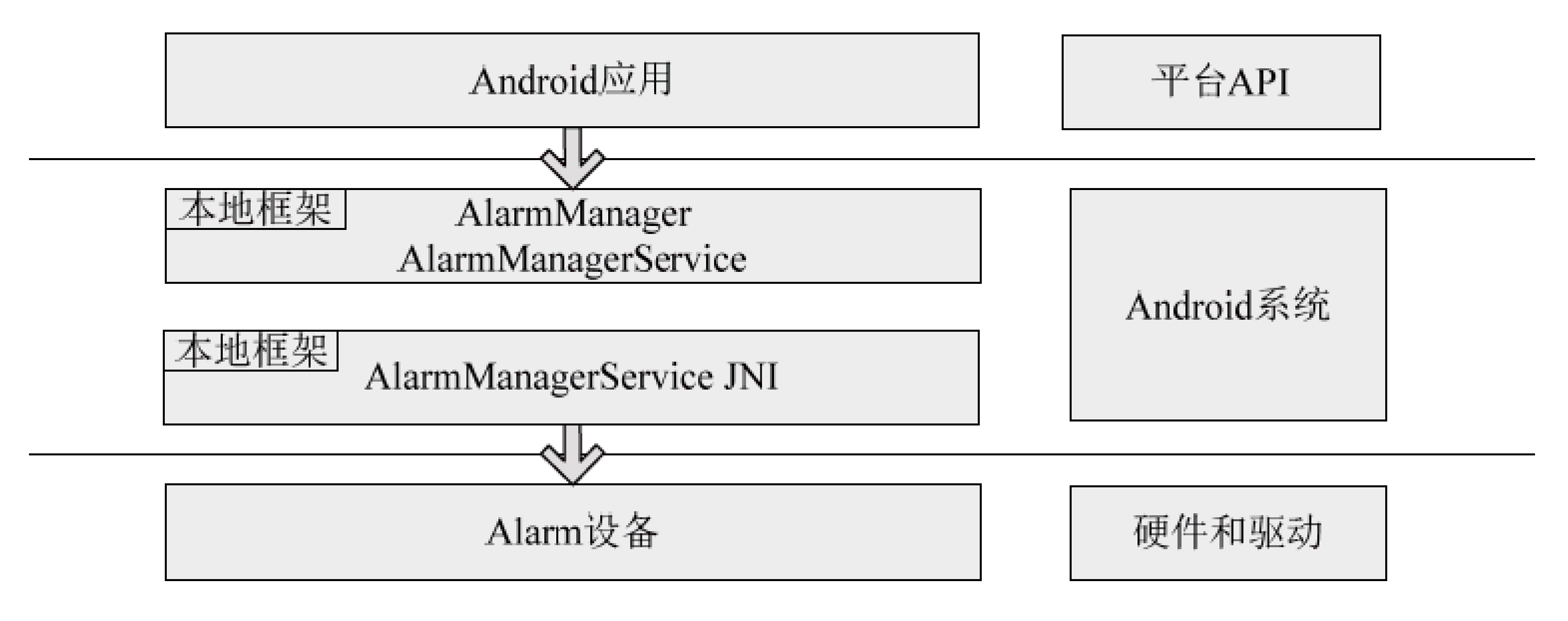


图 15-1　Alarm 系统的层次结构

由图 15-1 可知，Android 平台中 WiFi 系统从上到下主要包括：AlarmManager、AlarmManagerService Java、AlarmManagerService JNI、Alarm 驱动程序和实时时钟（RTC）驱动程序，这几部分的系统结构如图 15-2 所示。

图 15-2 中各个部分的具体说明如下。

（1）RTC 驱动程序

Linux 的 Alarm 驱动程序代码路径在内核的 drivers/rtc/目录下，各个硬件的具体实现不同。

（2）Alarm 驱动程序

这是 Android 特定内核的组件，能够调用 RTC 系统的功能，但是本身和硬件无关。Alarm 驱动程序的

实现文件如下。

- ☑ drivers/staging/android/alarm.c
- ☑ drivers/staging/android/android_alarm.h
- ☑ drivers/staging/android/alarm-dev.c

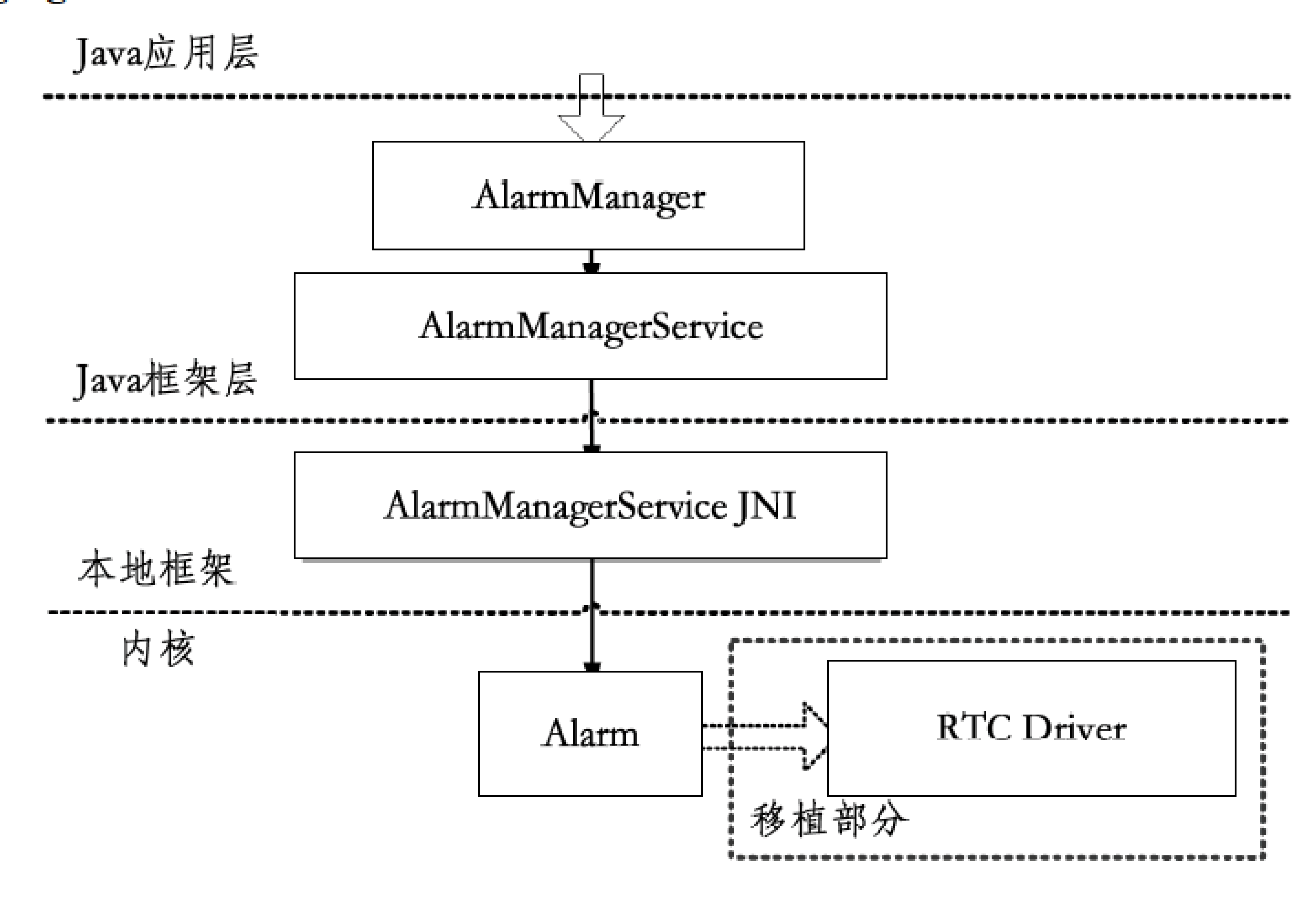


图 15-2　Alarm 的系统结构

（3）本地 JNI 部分

此部分的代码路径是 frameworks/base/services/jni/com_android_server_AlarmManagerService.cpp。

此文件是 Alarm 部分的本地代码，也同时提供了 JNI 的接口。

（4）Java 部分

此部分的代码路径是 frameworks/base/services/java/com/android/server/AlarmManagerSe rvice.java 和 frameworks/base/core/java/android/app/AlarmManager.java。

在文件 AlarmManagerService.java 中实现了 android.server 包中的 AlarmManagerService，在文件 AlarmManager.java 中实现了 android.app 包中的 AlarmManager 类，它通过使用 AlarmManagerService 服务实现，并对 Java 层提供了平台 API。

15.1.2　需要移植的内容

Android 的 Alarm 系统的 Java 层、本地部分的代码都是标准的，所以不需要更改。内核中的 Alarm 驱动程序与硬件无关，在 Android 系统中都是相同的。所以警报器系统的移植实际上就是 RTC 驱动程序的移植。RTC 驱动程序是 Linux 中一种标准的驱动程序，它在用户空间也提供了设备节点（自定义的字符设备或 MISC 字符设备）。根据 Android 系统的情况，不直接使用 RTC 驱动程序，而是通过 Alarm 驱动程序调用 RTC 系统，而 Android 系统的用户空间只调用 Alarm 驱动程序。

15.2　RTC 驱动程序架构

RTC 驱动程序 RTC 是 Linux 中标准的 Alarm 驱动程序框架，此驱动程序的框架内容在内核文件 inlcude/linux/rtc.h 中定义。首先在此文件中定义了如下两个函数，功能是分别实现注册和注销 RTC 设备，



具体代码如下。

```
extern struct rtc_device *rtc_device_register(const char *name,
                                              struct device *dev,
                                              const struct rtc_class_ops *ops,
                                              struct module *owner);
extern void rtc_device_unregister(struct rtc_device *rtc);
```

然后定义结构体 rtc_class_ops，具体实现代码如下。

```
struct rtc_class_ops {
        int (*open)(struct device *);
        void (*release)(struct device *);
        int (*ioctl)(struct device *, unsigned int, unsigned long);
        int (*read_time)(struct device *, struct rtc_time *);
        int (*set_time)(struct device *, struct rtc_time *);
        int (*read_alarm)(struct device *, struct rtc_wkalrm *);
        int (*set_alarm)(struct device *, struct rtc_wkalrm *);
        int (*proc)(struct device *, struct seq_file *);
        int (*set_mmss)(struct device *, unsigned long secs);
        int (*read_callback)(struct device *, int data);
        int (*alarm_irq_enable)(struct device *, unsigned int enabled);
 };
```

结构体 struct rtc_device 是在 RTC 驱动程序中使用的，是对 struct device 的扩展，其中也包含了 rtc_class_ops 结构。RTC 驱动程序的实现实际上就是实现了 rtc_class_ops 中的函数指针，主要包括时间和警报器这两方面的内容。

在用户空间中，可以通过 RTC 驱动程序的设备节点对其进行调试，调试的方法是通过 ioctl 命令实现的。这些命令是在文件 rtc.h 中定义，以 RTC 开头。例如下面就是 4 个命令。

```
#define RTC_ALM_SET        _IOW('p', 0x07, struct rtc_time)        /* 设置警报器时间 */
#define RTC_ALM_READ       _IOR('p', 0x08, struct rtc_time)        /* 读取警报器时间 */
#define RTC_RD_TIME        _IOR('p', 0x09, struct rtc_time)        /* 读取 RTC 时间 */
#define RTC_SET_TIME       _IOW('p', 0x0a, struct rtc_time)        /* 设置 RTC 时间 */
```

15.3　Alarm 驱动架构

在 Android 系统中，Alarm 驱动程序为用户空间提供了设备节点/dev/alarm，这是一个主设备号为 10 的 Misc 字符设备，并且其次设备号是动态生成的。Alarm 驱动程序由内核代码中的如下文件实现。

☑　drivers/staging/android/alarm.c
☑　drivers/staging/android/android_alarm.h
☑　drivers/staging/android/alarm-dev.c

本节将详细讲解上述文件的具体实现流程。

15.3.1　分析文件 android_alarm.h

头文件 android_alarm.h 提供了到用户空间的各 ioctl 命令接口，具体实现代码如下。

```
16 #ifndef _LINUX_ANDROID_ALARM_H
17 #define _LINUX_ANDROID_ALARM_H
18
```

```
#include <linux/ioctl.h>
#include <linux/time.h>
#include <linux/compat.h>

enum android_alarm_type {
        /*返回码的比特数或设置报警参数*/
        ANDROID_ALARM_RTC_WAKEUP,
        ANDROID_ALARM_RTC,
        ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP,
        ANDROID_ALARM_ELAPSED_REALTIME,
        ANDROID_ALARM_SYSTEMTIME,

        ANDROID_ALARM_TYPE_COUNT,

        /*返回码的比特数*/
        /* ANDROID_ALARM_TIME_CHANGE = 16 */
};

enum android_alarm_return_flags {
        ANDROID_ALARM_RTC_WAKEUP_MASK = 1U << ANDROID_ALARM_RTC_WAKEUP,
        ANDROID_ALARM_RTC_MASK = 1U << ANDROID_ALARM_RTC,
        ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP_MASK =
                        1U << ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP,
        ANDROID_ALARM_ELAPSED_REALTIME_MASK =
                        1U << ANDROID_ALARM_ELAPSED_REALTIME,
        ANDROID_ALARM_SYSTEMTIME_MASK = 1U << ANDROID_ALARM_SYSTEMTIME,
        ANDROID_ALARM_TIME_CHANGE_MASK = 1U << 16
};

/*禁用报警*/
#define ANDROID_ALARM_CLEAR(type)           _IO('a', 0 | ((type) <<4))

/*确认最后报警并等待下一个*/
#define ANDROID_ALARM_WAIT                  _IO('a', 1)

#define ALARM_IOW(c, type, size)            _IOW('a', (c) | ((type) <<4), size)
/*设置报警*/
#define ANDROID_ALARM_SET(type)             ALARM_IOW(2, type, struct timespec)
#define ANDROID_ALARM_SET_AND_WAIT(type)    ALARM_IOW(3, type, struct timespec)
#define ANDROID_ALARM_GET_TIME(type)        ALARM_IOW(4, type, struct timespec)
#define ANDROID_ALARM_SET_RTC               _IOW('a',5, struct timespec)
#define ANDROID_ALARM_BASE_CMD(cmd)         (cmd & ~(_IOC(0, 0, 0xf0, 0)))
#define ANDROID_ALARM_IOCTL_TO_TYPE(cmd)    (_IOC_NR(cmd) >>4)


#ifdef CONFIG_COMPAT
#define ANDROID_ALARM_SET_COMPAT(type)              ALARM_IOW(2, type, \
                                                    struct compat_timespec)
#define ANDROID_ALARM_SET_AND_WAIT_COMPAT(type) ALARM_IOW(3, type, \
                                                    struct compat_timespec)
#define ANDROID_ALARM_GET_TIME_COMPAT(type)         ALARM_IOW(4, type, \
```

```
                                                        struct compat_timespec)
#define ANDROID_ALARM_SET_RTC_COMPAT                     _IOW('a',5, \
                                                        struct compat_timespec)
#define ANDROID_ALARM_IOCTL_NR(cmd)                    (_IOC_NR(cmd) & ((1<<4)-1))
#define ANDROID_ALARM_COMPAT_TO_NORM(cmd)   \
                              ALARM_IOW(ANDROID_ALARM_IOCTL_NR(cmd), \
                                    ANDROID_ALARM_IOCTL_TO_TYPE(cmd), \
                                    struct timespec)

#endif

#endif
```

上述代码的核心是枚举 android_alarm_type，在里面定义了一些和 Alarm 相关的信息，主要包括如下 5 种类型的 Alarm。

☑ ANDROID_RTC_WAKEUP 类型：表示在触发 Alarm 时需要唤醒设备，反之则不需要唤醒设备。

☑ ANDROID_ALARM_RTC_WAKEUP 类型：表示在指定的某一时刻触发 Alarm。

☑ ANDROID_ALARM_ELAPSED_REALTIME 类型：表示在设备启动后，流逝的时间达到总时间之后触发 Alarm。

☑ ANDROID_ALARM_SYSTEMTIME 类型：表示系统时间。

☑ ANDROID_ALARM_TYPE_COUNT 类型：表示 Alram 类型的计数。

Alarm 返回标记随着 Alarm 的类型而改变。通过定义的宏实现禁用 Alarm、Alarm 等待、设置 Alarm 等功能。

15.3.2　分析文件 alarm.c

在 Android 系统中，Alarm 模块提供了如下两个设备。

☑ Alarm 的 Platform Driver，实现文件是 alarm.c。

☑ 暴露给用户使用的接口 Misc 的 Alarm 接口，实现文件是 alarm-dev.c。

文件 alarm.c 的功能是定义一系列的 ioctl 函数，来操纵 Platform Alarm Driver 提供的功能。本节将首先讲解文件 alarm.c 的具体实现源码。

（1）进行初始化处理，函数 alarm_driver_init()的功能是初始化 5 个 Alarm Device 相关联的 hrtimer 定时器，设置 hrtimer 定时器的回调函数为 alarm_timer_triggered()，然后再注册一个 Plateform Driver 和 class interface。函数 alarm_driver_init()的具体实现代码如下。

```
static int __init alarm_driver_init(void)
{
        int err;
        int i;

        for (i = 0; i < ANDROID_ALARM_SYSTEMTIME; i++) {
                hrtimer_init(&alarms[i].timer,
                                CLOCK_REALTIME, HRTIMER_MODE_ABS);
                alarms[i].timer.function = alarm_timer_triggered;
        }
        hrtimer_init(&alarms[ANDROID_ALARM_SYSTEMTIME].timer,
                        CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
        alarms[ANDROID_ALARM_SYSTEMTIME].timer.function = alarm_timer_triggered;
        err = platform_driver_register(&alarm_driver);
```

```
	if (err < 0)
		goto err1;
	wake_lock_init(&alarm_rtc_wake_lock, WAKE_LOCK_SUSPEND, "alarm_rtc");
	rtc_alarm_interface.class = rtc_class;
	err = class_interface_register(&rtc_alarm_interface);
	if (err < 0)
		goto err2;

	return 0;

err2:
	wake_lock_destroy(&alarm_rtc_wake_lock);
	platform_driver_unregister(&alarm_driver);
err1:
	return err;
}
```

在上述代码中用到了 platform_driver 类型的 alarm_driver 结构体，具体实现代码如下。

```
static struct platform_driver alarm_driver = {
	.suspend = alarm_suspend,
	.resume = alarm_resume,
	.driver = {
		.name = "alarm"
	}
};
```

在上述代码中，指定了当系统挂起（suspend）和唤醒（Desume）时所需要的实现，分别是 alarm_suspend 和 alarm_resume，同时将 Alarm 设备驱动的名称设置为 alarm。

函数 alarm_suspend()的具体实现代码如下。

```
static int alarm_suspend(struct platform_device *pdev, pm_message_t state)
{
	int                     err = 0;
	unsigned long           flags;
	struct rtc_wkalrm       rtc_alarm;
	struct rtc_time         rtc_current_rtc_time;
	unsigned long           rtc_current_time;
	unsigned long           rtc_alarm_time;
	struct timespec         rtc_delta;
	struct timespec         wall_time;
	struct alarm_queue *wakeup_queue = NULL;
	struct alarm_queue *tmp_queue = NULL;

	pr_alarm(SUSPEND, "alarm_suspend(%p, %d)\n", pdev, state.event);

	spin_lock_irqsave(&alarm_slock, flags);
	suspended = true;
	spin_unlock_irqrestore(&alarm_slock, flags);

	hrtimer_cancel(&alarms[ANDROID_ALARM_RTC_WAKEUP].timer);
	hrtimer_cancel(&alarms[
			ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP].timer);

```

```
		tmp_queue = &alarms[ANDROID_ALARM_RTC_WAKEUP];
		if (tmp_queue->first)
			wakeup_queue = tmp_queue;
		tmp_queue = &alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP];
		if (tmp_queue->first && (!wakeup_queue ||
					hrtimer_get_expires(&tmp_queue->timer).tv64 <
					hrtimer_get_expires(&wakeup_queue->timer).tv64))
			wakeup_queue = tmp_queue;
		if (wakeup_queue) {
			rtc_read_time(alarm_rtc_dev, &rtc_current_rtc_time);
			getnstimeofday(&wall_time);
			rtc_tm_to_time(&rtc_current_rtc_time, &rtc_current_time);
			set_normalized_timespec(&rtc_delta,
						wall_time.tv_sec - rtc_current_time,
						wall_time.tv_nsec);

			rtc_alarm_time = timespec_sub(ktime_to_timespec(
				hrtimer_get_expires(&wakeup_queue->timer)),
				rtc_delta).tv_sec;

			rtc_time_to_tm(rtc_alarm_time, &rtc_alarm.time);
			rtc_alarm.enabled = 1;
			rtc_set_alarm(alarm_rtc_dev, &rtc_alarm);
			rtc_read_time(alarm_rtc_dev, &rtc_current_rtc_time);
			rtc_tm_to_time(&rtc_current_rtc_time, &rtc_current_time);
			pr_alarm(SUSPEND,
				"rtc alarm set at %ld, now %ld, rtc delta %ld.%09ld\n",
				rtc_alarm_time, rtc_current_time,
				rtc_delta.tv_sec, rtc_delta.tv_nsec);
			if (rtc_current_time + 1 >= rtc_alarm_time) {
				pr_alarm(SUSPEND, "alarm about to go off\n");
				memset(&rtc_alarm, 0, sizeof(rtc_alarm));
				rtc_alarm.enabled = 0;
				rtc_set_alarm(alarm_rtc_dev, &rtc_alarm);

				spin_lock_irqsave(&alarm_slock, flags);
				suspended = false;
				wake_lock_timeout(&alarm_rtc_wake_lock, 2 * HZ);
				update_timer_locked(&alarms[ANDROID_ALARM_RTC_WAKEUP],
										false);
				update_timer_locked(&alarms[
					ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP], false);
				err = -EBUSY;
				spin_unlock_irqrestore(&alarm_slock, flags);
			}
		}
		return err;
}
```

函数 alarm_resume()的具体实现代码如下。

```
static int alarm_resume(struct platform_device *pdev)
{
```

```
	struct rtc_wkalrm alarm;
	unsigned long		flags;

	pr_alarm(SUSPEND, "alarm_resume(%p)\n", pdev);

	memset(&alarm, 0, sizeof(alarm));
	alarm.enabled = 0;
	rtc_set_alarm(alarm_rtc_dev, &alarm);

	spin_lock_irqsave(&alarm_slock, flags);
	suspended = false;
	update_timer_locked(&alarms[ANDROID_ALARM_RTC_WAKEUP], false);
	update_timer_locked(&alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP],
								false);
	spin_unlock_irqrestore(&alarm_slock, flags);

	return 0;
}
```

另外，在函数 alarm_driver_init()中还用到了类 class_interface 的接口 rtc_alarm_interface，具体实现代码如下。

```
static struct class_interface rtc_alarm_interface = {
	.add_dev = &rtc_alarm_add_device,
	.remove_dev = &rtc_alarm_remove_device,
};
```

（2）定时器回调函数 alarm_timer_triggered()的功能是轮询红黑树中所有的 Alarm 节点中符合条件的节点，如果有，则执行 alarm.functionalarm 指向文件 alarm_dev.c 中的 alarm_triggered()函数。这是因为在执行文件 alarm_dev.c 中的 alarm_init()函数时，会把每个 Alarm 节点的 function 设置成 alarm_triggered。具体实现代码如下。

```
static enum hrtimer_restart alarm_timer_triggered(struct hrtimer *timer)
{
	struct alarm_queue *base;
	struct android_alarm *alarm;
	unsigned long flags;
	ktime_t now;

	spin_lock_irqsave(&alarm_slock, flags);

	base = container_of(timer, struct alarm_queue, timer);
	now = base->stopped ? base->stopped_time : hrtimer_cb_get_time(timer);
	now = ktime_sub(now, base->delta);

	pr_alarm(INT, "alarm_timer_triggered type %ld at %lld\n",
		base - alarms, ktime_to_ns(now));

	while (base->first) {
		alarm = container_of(base->first, struct android_alarm, node);
		if (alarm->softexpires.tv64 > now.tv64) {
			pr_alarm(FLOW, "don't call alarm, %pF, %lld (s %lld)\n",
				alarm->function, ktime_to_ns(alarm->expires),
```

```
                        ktime_to_ns(alarm->softexpires));
                break;
            }
            base->first = rb_next(&alarm->node);
            rb_erase(&alarm->node, &base->alarms);
            RB_CLEAR_NODE(&alarm->node);
            pr_alarm(CALL, "call alarm, type %d, func %pF, %lld (s %lld)\n",
                alarm->type, alarm->function,
                ktime_to_ns(alarm->expires),
                ktime_to_ns(alarm->softexpires));
            spin_unlock_irqrestore(&alarm_slock, flags);
            alarm->function(alarm);
            spin_lock_irqsave(&alarm_slock, flags);
        }
        if (!base->first)
            pr_alarm(FLOW, "no more alarms of type %ld\n", base - alarms);
        update_timer_locked(base, true);
        spin_unlock_irqrestore(&alarm_slock, flags);
        return HRTIMER_NORESTART;
}
```

在上述代码中，用到了函数 alarm_triggered()和 alarm_timer_triggered()。其中前者是 rtc 芯片的 Alarm 中断的回调函数，后者是 android alarm_queue_gttimer 到时的回调函数。

（3）函数 rtc_alarm_add_device()的功能是注册一个 rtc 以中断 rtc_irq_register，具体实现代码如下。

```
static int rtc_alarm_add_device(struct device *dev,
                        struct class_interface *class_intf)
{
        int err;
        struct rtc_device *rtc = to_rtc_device(dev);

        mutex_lock(&alarm_setrtc_mutex);

        if (alarm_rtc_dev) {
            err = -EBUSY;
            goto err1;
        }

        alarm_platform_dev =
            platform_device_register_simple("alarm", -1, NULL, 0);
        if (IS_ERR(alarm_platform_dev)) {
            err = PTR_ERR(alarm_platform_dev);
            goto err2;
        }
        err = rtc_irq_register(rtc, &alarm_rtc_task);
        if (err)
            goto err3;
        alarm_rtc_dev = rtc;
        pr_alarm(INIT_STATUS, "using rtc device, %s, for alarms", rtc->name);
        mutex_unlock(&alarm_setrtc_mutex);

        return 0;
```

```

err3:
        platform_device_unregister(alarm_platform_dev);
err2:
err1:
        mutex_unlock(&alarm_setrtc_mutex);
        return err;
}
```

（4）中断回调函数 alarm_triggered_func()的具体实现代码如下。

```
static void alarm_triggered_func(void *p)
{
        struct rtc_device *rtc = alarm_rtc_dev;
        if (!(rtc->irq_data & RTC_AF))
                return;
        pr_alarm(INT, "rtc alarm triggered\n");
        wake_lock_timeout(&alarm_rtc_wake_lock, 1 * HZ);
}
```

通过上述实现代码可知，当硬件 rtc chip 中的 Alarm 发生中断时，系统会调用函数 alarm_triggered_func()。函数 alarm_triggered_func()的功能是调用函数 wake_lock_timeout()锁住 alarm_rtc_wake_lock 1 秒。因为这时 Alarm 会进入 alarm_resumelock 锁住 alarm_rtc_wake_lock，这样可以防止 Alarm 在此时进入 suspend 状态。

（5）再来分析唤醒和休眠，Wakelock 有加锁和解锁两种操作，而加锁又可以分为如下两种方式。

- ☑ 永久加锁（wake_lock）：这种锁必须手动解锁。
- ☑ 超时锁（wake_lock_timeout）：这种锁在超过指定时间后，会自动解锁。

永久加锁（wake_lock）和超时锁（wake_lock_timeout）的实现代码如下。

```
void wake_lock(struct wake_lock *lock)
{
wake_lock_internal(lock, 0, 0);
}
void wake_lock_timeout(struct wake_lock *lock, long timeout)
{
wake_lock_internal(lock, timeout, 1);
}
```

对于 wakelock 来说，如果 timeout=has_timeout=0，则直接加锁后退出。在上述代码中用到了函数 wake_lock_internal()，具体实现代码如下。

```
static void wake_lock_internal(
struct wake_lock *lock, long timeout, int has_timeout)
{
int type;
unsigned long irqflags;
long expire_in;
spin_lock_irqsave(&list_lock, irqflags);
type = lock->flags & WAKE_LOCK_TYPE_MASK;
BUG_ON(type >= WAKE_LOCK_TYPE_COUNT);
BUG_ON(!(lock->flags & WAKE_LOCK_INITIALIZED));
#ifdef CONFIG_WAKELOCK_STAT
if (type == WAKE_LOCK_SUSPEND && wait_for_wakeup) {
if (debug_mask & DEBUG_WAKEUP)
pr_info("wakeup wake lock: %s\n", lock->name);
wait_for_wakeup = 0;
```

```
lock->stat.wakeup_count++;
}
if ((lock->flags & WAKE_LOCK_AUTO_EXPIRE) &&
    (long)(lock->expires - jiffies) <= 0) {
wake_unlock_stat_locked(lock, 0);
lock->stat.last_time = ktime_get();
}
#endif
if (!(lock->flags & WAKE_LOCK_ACTIVE)) {
lock->flags |= WAKE_LOCK_ACTIVE;
#ifdef CONFIG_WAKELOCK_STAT
lock->stat.last_time = ktime_get();
#endif
}
list_del(&lock->link);
if (has_timeout) {
if (debug_mask & DEBUG_WAKE_LOCK)
pr_info("wake_lock: %s, type %d, timeout %ld.%03lu\n",
lock->name, type, timeout / HZ,
(timeout % HZ) * MSEC_PER_SEC / HZ);
lock->expires = jiffies + timeout;
lock->flags |= WAKE_LOCK_AUTO_EXPIRE;
list_add_tail(&lock->link, &active_wake_locks[type]);
} else {
if (debug_mask & DEBUG_WAKE_LOCK)
pr_info("wake_lock: %s, type %d\n", lock->name, type);
lock->expires = LONG_MAX;
lock->flags &= ~WAKE_LOCK_AUTO_EXPIRE;
list_add(&lock->link, &active_wake_locks[type]);
}
if (type == WAKE_LOCK_SUSPEND) {
current_event_num++;
#ifdef CONFIG_WAKELOCK_STAT
if (lock == &main_wake_lock)
update_sleep_wait_stats_locked(1);
else if (!wake_lock_active(&main_wake_lock))
update_sleep_wait_stats_locked(0);
#endif
if (has_timeout)
expire_in = has_wake_lock_locked(type);
else
expire_in = -1;
if (expire_in > 0) {
if (debug_mask & DEBUG_EXPIRE)
pr_info("wake_lock: %s, start expire timer, "
"%ld\n", lock->name, expire_in);
mod_timer(&expire_timer, jiffies + expire_in);
} else {
if (del_timer(&expire_timer))
if (debug_mask & DEBUG_EXPIRE)
pr_info("wake_lock: %s, stop expire timer\n",
```

```
lock->name);
if (expire_in == 0)
queue_work(suspend_work_queue, &suspend_work);
}
}
spin_unlock_irqrestore(&list_lock, irqflags);
}
```

而对于函数 wake_lock_timeout()来说，在经过 timeout 时间后才可以加锁。当判断当前持有 wakelock 时启动另一个定时器，然后在 expire_timer 的回调函数中再次判断是否持有 wakelock。函数 expire_wake_locks()的具体实现代码如下所示。

```
static void expire_wake_locks(unsigned long data)
{
long has_lock;
unsigned long irqflags;
if (debug_mask & DEBUG_EXPIRE)
pr_info("expire_wake_locks: start\n");
spin_lock_irqsave(&list_lock, irqflags);
if (debug_mask & DEBUG_SUSPEND)
print_active_locks(WAKE_LOCK_SUSPEND);
has_lock = has_wake_lock_locked(WAKE_LOCK_SUSPEND);
if (debug_mask & DEBUG_EXPIRE)
pr_info("expire_wake_locks: done, has_lock %ld\n", has_lock);
if (has_lock == 0)
queue_work(suspend_work_queue, &suspend_work);
spin_unlock_irqrestore(&list_lock, irqflags);
}
```

在 wakelock 中，有如下两个地方可以让系统从 early_suspend 进入 suspend 状态。

- ☑ 在 wake_unlock 中，解锁之后，若没有其他的 wakelock，则进入 suspend。
- ☑ 在超时锁的定时器超时后，定时器的回调函数会判断有没有其他的 wakelock，如果没有，则进入 suspend。

（6）再看函数 alarm_late_init()，当启动 Alarm 后需要读取当前的 RCT 和系统时间，由于需要确保在这个操作过程中不被中断，或者在中断之后能告诉其他进程该过程没有读取完成，不能被请求，因此这里需要通过函数 spin_lock_irqsave()和 spin_unlock_irqrestore()来对其执行锁定和解锁操作。函数 alarm_late_init()的具体实现代码如下。

```
static int __init alarm_late_init(void)
{
	unsigned long	flags;
	struct timespec tmp_time, system_time;

	/* this needs to run after the rtc is read at boot */
	spin_lock_irqsave(&alarm_slock, flags);
	/* We read the current rtc and system time so we can later calulate
	 * elasped realtime to be (boot_systemtime + rtc - boot_rtc) ==
	 * (rtc - (boot_rtc - boot_systemtime))
	 */
	getnstimeofday(&tmp_time);
	ktime_get_ts(&system_time);
	alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP].delta =
```

```
                    alarms[ANDROID_ALARM_ELAPSED_REALTIME].delta =
                            timespec_to_ktime(timespec_sub(tmp_time, system_time));

            spin_unlock_irqrestore(&alarm_slock, flags);
            return 0;
}
```

（7）再看函数 alarm_exit()，当 Alarm 退出时，需要通过函数 class_interface_unregister()卸载在初始化时注册的 Alarm 接口，通过 wake_lock_destroy()函数销毁 SUSPEND lock，并通过函数 platform_ driver_unregister()卸载 Alarm 驱动。函数 alarm_exit()的具体实现代码如下。

```
static void __exit alarm_exit(void) {
class_interface_unregister(&rtc_alarm_interface);
wake_lock_destroy(&alarm_rtc_wake_lock);
wake_lock_destroy(&alarm_wake_lock);
platform_driver_unregister(&alarm_driver);
}
```

（8）接下来讲解函数 rtc_alarm_add_device()和 rtc_alarm_remove_device()的具体实现。在添加设备时，首先将设备转换成 rtc_device 类型，然后通过函数 misc_register()将自己注册成为一个 Misc 设备。在此功能阶段的主要对应代码如下。

```
static struct file_operations alarm_fops = {
.owner = THIS_MODULE,
.unlocked_ioctl = alarm_ioctl,
.open = alarm_open,
.release = alarm_release,
};
static struct miscdevice alarm_device = {
.minor = MISC_DYNAMIC_MINOR,
.name = "alarm",
.fops = &alarm_fops,
};
```

在上述代码中，alarm_device 中的.name 表示设备文件名称，alarm_fops 定义了 Alarm 的常用操作，包括打开、释放和 I/O 控制。另外还需要通过 rtc_irq_register()函数注册一个 rtc_task，用来处理 Alarm 触发的方法。

（9）函数 android_alarm_set_rtc()的功能是处理在指定的某一时刻触发 Alarm 的事件，具体实现代码如下。

```
int android_alarm_set_rtc(struct timespec new_time)
{
            int i;
            int ret;
            unsigned long flags;
            struct rtc_time rtc_new_rtc_time;
            struct timespec tmp_time;

            rtc_time_to_tm(new_time.tv_sec, &rtc_new_rtc_time);

            pr_alarm(TSET, "set rtc %ld %ld - rtc %02d:%02d:%02d %02d/%02d/%04d\n",
                    new_time.tv_sec, new_time.tv_nsec,
                    rtc_new_rtc_time.tm_hour, rtc_new_rtc_time.tm_min,
                    rtc_new_rtc_time.tm_sec, rtc_new_rtc_time.tm_mon + 1,
                    rtc_new_rtc_time.tm_mday,
                    rtc_new_rtc_time.tm_year + 1900);
```

```

	mutex_lock(&alarm_setrtc_mutex);
	spin_lock_irqsave(&alarm_slock, flags);
	wake_lock(&alarm_rtc_wake_lock);
	getnstimeofday(&tmp_time);
	for (i = 0; i < ANDROID_ALARM_SYSTEMTIME; i++) {
		hrtimer_try_to_cancel(&alarms[i].timer);
		alarms[i].stopped = true;
		alarms[i].stopped_time = timespec_to_ktime(tmp_time);
	}
	alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP].delta =
		alarms[ANDROID_ALARM_ELAPSED_REALTIME].delta =
		ktime_sub(alarms[ANDROID_ALARM_ELAPSED_REALTIME].delta,
			timespec_to_ktime(timespec_sub(tmp_time, new_time)));
	spin_unlock_irqrestore(&alarm_slock, flags);
	ret = do_settimeofday(&new_time);
	spin_lock_irqsave(&alarm_slock, flags);
	for (i = 0; i < ANDROID_ALARM_SYSTEMTIME; i++) {
		alarms[i].stopped = false;
		update_timer_locked(&alarms[i], false);
	}
	spin_unlock_irqrestore(&alarm_slock, flags);
	if (ret < 0) {
		pr_alarm(ERROR, "alarm_set_rtc: Failed to set time\n");
		goto err;
	}
	if (!alarm_rtc_dev) {
		pr_alarm(ERROR,
			"alarm_set_rtc: no RTC, time will be lost on reboot\n");
		goto err;
	}
	ret = rtc_set_time(alarm_rtc_dev, &rtc_new_rtc_time);
	if (ret < 0)
		pr_alarm(ERROR, "alarm_set_rtc: "
			"Failed to set RTC, time will be lost on reboot\n");
err:
	wake_unlock(&alarm_rtc_wake_lock);
	mutex_unlock(&alarm_setrtc_mutex);
	return ret;
}
```

（10）函数 android_alarm_cancel()的功能是取消报警功能并等待处理完成，返回 0 表示报警是不活跃的，返回 1 表示报警是活跃的。函数 android_alarm_cancel()的具体实现代码如下。

```
int android_alarm_cancel(struct android_alarm *alarm)
{
	for (;;) {
		int ret = android_alarm_try_to_cancel(alarm);
		if (ret >= 0)
			return ret;
		cpu_relax();
	}
}
```

（11）函数 android_alarm_try_to_cancel()的功能是停止报警功能，具体实现代码如下。

```
int android_alarm_try_to_cancel(struct android_alarm *alarm)
{
        struct alarm_queue *base = &alarms[alarm->type];
        unsigned long flags;
        bool first = false;
        int ret = 0;

        spin_lock_irqsave(&alarm_slock, flags);
        if (!RB_EMPTY_NODE(&alarm->node)) {
                pr_alarm(FLOW, "canceled alarm, type %d, func %pF at %lld\n",
                        alarm->type, alarm->function,
                        ktime_to_ns(alarm->expires));
                ret = 1;
                if (base->first == &alarm->node) {
                        base->first = rb_next(&alarm->node);
                        first = true;
                }
                rb_erase(&alarm->node, &base->alarms);
                RB_CLEAR_NODE(&alarm->node);
                if (first)
                        update_timer_locked(base, true);
        } else
                pr_alarm(FLOW, "tried to cancel alarm, type %d, func %pF\n",
                        alarm->type, alarm->function);
        spin_unlock_irqrestore(&alarm_slock, flags);
        if (!ret && hrtimer_callback_running(&base->timer))
                ret = -1;
        return ret;
}
```

（12）函数 alarm_enqueue_locked()用于在 Alarm 初始化函数中注册 hrtimer 定时器码，其回调函数是 alarm_timer_triggered()，但是 hrtimer 定时器和 Alarm 只是进行了初始化工作，只有在 hrtimer_start 后才能使用 hrtimer 定时器。函数 update_timer_locked()是在函数 alarm_enqueue_locked()中被调用的，此函数的具体实现代码如下。

```
static void alarm_enqueue_locked(struct android_alarm *alarm)
{
        struct alarm_queue *base = &alarms[alarm->type];
        struct rb_node **link = &base->alarms.rb_node;
        struct rb_node *parent = NULL;
        struct android_alarm *entry;
        int leftmost = 1;
        bool was_first = false;

        pr_alarm(FLOW, "added alarm, type %d, func %pF at %lld\n",
                alarm->type, alarm->function, ktime_to_ns(alarm->expires));

        if (base->first == &alarm->node) {
                base->first = rb_next(&alarm->node);
                was_first = true;
        }
```

```
        if (!RB_EMPTY_NODE(&alarm->node)) {
                rb_erase(&alarm->node, &base->alarms);
                RB_CLEAR_NODE(&alarm->node);
        }

        while (*link) {
                parent = *link;
                entry = rb_entry(parent, struct android_alarm, node);
                /*
                * We dont care about collisions. Nodes with
                * the same expiry time stay together.
                */
                if (alarm->expires.tv64 < entry->expires.tv64) {
                        link = &(*link)->rb_left;
                } else {
                        link = &(*link)->rb_right;
                        leftmost = 0;
                }
        }
        if (leftmost)
                base->first = &alarm->node;
        if (leftmost || was_first)
                update_timer_locked(base, was_first);

        rb_link_node(&alarm->node, parent, link);
        rb_insert_color(&alarm->node, &base->alarms);
}
```

（13）函数 update_timer_locked()的功能是修改处理报警，具体实现代码如下。

```
static void update_timer_locked(struct alarm_queue *base, bool head_removed)
{
        struct android_alarm *alarm;
        bool is_wakeup = base == &alarms[ANDROID_ALARM_RTC_WAKEUP] ||
                        base == &alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP];

        if (base->stopped) {
                pr_alarm(FLOW, "changed alarm while setting the wall time\n");
                return;
        }

        if (is_wakeup && !suspended && head_removed)
                wake_unlock(&alarm_rtc_wake_lock);

        if (!base->first)
                return;

        alarm = container_of(base->first, struct android_alarm, node);

        pr_alarm(FLOW, "selected alarm, type %d, func %pF at %lld\n",
                alarm->type, alarm->function, ktime_to_ns(alarm->expires));

        if (is_wakeup && suspended) {
```

```
                        pr_alarm(FLOW, "changed alarm while suspened\n");
                        wake_lock_timeout(&alarm_rtc_wake_lock, 1 * HZ);
                        return;
            }

            hrtimer_try_to_cancel(&base->timer);
            base->timer.node.expires = ktime_add(base->delta, alarm->expires);
            base->timer._softexpires = ktime_add(base->delta, alarm->softexpires);
            hrtimer_start_expires(&base->timer, HRTIMER_MODE_ABS);
}
```

15.3.3　分析文件 alarm-dev.c

在 Android 系统中，文件 alarm-dev.c 以文件 alarm.c 为基础实现了与应用层的交互功能，暴露了 miscdevice 的设备接口。文件 alarm-dev.c 的具体实现流程如下。

（1）定义如下所示的全局变量。

```
//标志位，表示 Alarm 设备是否被打开
static int alarm_opened;
static DEFINE_SPINLOCK(alarm_slock);
static struct wakeup_source alarm_wake_lock;
static DECLARE_WAIT_QUEUE_HEAD(alarm_wait_queue);
//表示设备是否就绪
static uint32_t alarm_pending;
static uint32_t alarm_enabled;
static uint32_t wait_pending;
//每种类型一个 Alarm 设备，Android 目前创建了 5 个 Alarm 设备
static struct devalarm alarms[ANDROID_ALARM_TYPE_COUNT];
```

（2）分别定义模块初始化函数 alarm_dev_init()和 exit 函数 alarm_dev_exit()，具体实现代码如下。

```
static int __init alarm_dev_init(void)
{
        int err;
        int i;

        err = misc_register(&alarm_device);
        if (err)
                return err;

        alarm_init(&alarms[ANDROID_ALARM_RTC_WAKEUP].u.alrm,
                        ALARM_REALTIME, devalarm_alarmhandler);
        hrtimer_init(&alarms[ANDROID_ALARM_RTC].u.hrt,
                        CLOCK_REALTIME, HRTIMER_MODE_ABS);
        alarm_init(&alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP].u.alrm,
                        ALARM_BOOTTIME, devalarm_alarmhandler);
        hrtimer_init(&alarms[ANDROID_ALARM_ELAPSED_REALTIME].u.hrt,
                        CLOCK_BOOTTIME, HRTIMER_MODE_ABS);
        hrtimer_init(&alarms[ANDROID_ALARM_SYSTEMTIME].u.hrt,
                        CLOCK_MONOTONIC, HRTIMER_MODE_ABS);

        for (i = 0; i < ANDROID_ALARM_TYPE_COUNT; i++) {
```

```
				alarms[i].type = i;
				if (!is_wakeup(i))
					alarms[i].u.hrt.function = devalarm_hrthandler;
		}

		wakeup_source_init(&alarm_wake_lock, "alarm");
		return 0;
}

static void __exit alarm_dev_exit(void)
{
		misc_deregister(&alarm_device);
		wakeup_source_trash(&alarm_wake_lock);
}

module_init(alarm_dev_init);
module_exit(alarm_dev_exit);
```

通过上述实现代码可知，初始化函数 alarm_dev_init 的功能是调用函数 misc_register()注册一个 miscdevice，其中定义报警器设备的结构体的代码如下。

```
static struct miscdevice alarm_device = {
		.minor = MISC_DYNAMIC_MINOR,
		.name = "alarm",
		.fops = &alarm_fops,
};
```

对应的 File Operations 为 alarm_fops，具体实现代码如下。

```
static const struct file_operations alarm_fops = {
		.owner = THIS_MODULE,
		.unlocked_ioctl = alarm_ioctl,
		.open = alarm_open,
		.release = alarm_release,
#ifdef CONFIG_COMPAT
		.compat_ioctl = alarm_compat_ioctl,
#endif
};
```

接下来需要为每个 alarm device 调用初始化函数 alarm_init()，此函数的代码在文件 alarm.c 中实现。

（3）再看模块 Misc Device 的标准接口函数 alarm_open()、alarm_release()和 alarm_ioctl()，具体实现代码如下。

```
static int alarm_open(struct inode *inode, struct file *file)
{
		file->private_data = NULL;
		return 0;
}

static int alarm_release(struct inode *inode, struct file *file)
{
		int i;
		unsigned long flags;

		spin_lock_irqsave(&alarm_slock, flags);
		if (file->private_data) {
```

```
		for (i = 0; i < ANDROID_ALARM_TYPE_COUNT; i++) {
			uint32_t alarm_type_mask = 1U << i;
			if (alarm_enabled & alarm_type_mask) {
				alarm_dbg(INFO,
					"%s: clear alarm, pending %d\n",
					__func__,
					!!(alarm_pending & alarm_type_mask));
				alarm_enabled &= ~alarm_type_mask;
			}
			spin_unlock_irqrestore(&alarm_slock, flags);
			devalarm_cancel(&alarms[i]);
			spin_lock_irqsave(&alarm_slock, flags);
		}
		if (alarm_pending | wait_pending) {
			if (alarm_pending)
				alarm_dbg(INFO, "%s: clear pending alarms %x\n",
					__func__, alarm_pending);
			__pm_relax(&alarm_wake_lock);
			wait_pending = 0;
			alarm_pending = 0;
		}
		alarm_opened = 0;
	}
	spin_unlock_irqrestore(&alarm_slock, flags);
	return 0;
}
static long alarm_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{

	struct timespec ts;
	int rv;

	switch (ANDROID_ALARM_BASE_CMD(cmd)) {
	case ANDROID_ALARM_SET_AND_WAIT(0):
	case ANDROID_ALARM_SET(0):
	case ANDROID_ALARM_SET_RTC:
		if (copy_from_user(&ts, (void __user *)arg, sizeof(ts)))
			return -EFAULT;
		break;
	}

	rv = alarm_do_ioctl(file, cmd, &ts);
	if (rv)
		return rv;

	switch (ANDROID_ALARM_BASE_CMD(cmd)) {
	case ANDROID_ALARM_GET_TIME(0):
		if (copy_to_user((void __user *)arg, &ts, sizeof(ts)))
			return -EFAULT;
		break;
	}
```

```

	return 0;
}
```

（4）当 Alarm 定时时间到时会调用函数 devalarm_triggered()，当定时闹铃的时间到了之后，函数 alarm_timer_triggered()会调用该毁掉函数。函数 devalarm_triggered()的具体实现代码如下。

```
static void devalarm_triggered(struct devalarm *alarm)
{
	unsigned long flags;
	uint32_t alarm_type_mask = 1U << alarm->type;

	alarm_dbg(INT, "%s: type %d\n", __func__, alarm->type);
	spin_lock_irqsave(&alarm_slock, flags);
	if (alarm_enabled & alarm_type_mask) {
		__pm_wakeup_event(&alarm_wake_lock, 5000); /* 5secs */
		alarm_enabled &= ~alarm_type_mask;
		alarm_pending |= alarm_type_mask;
		wake_up(&alarm_wait_queue);
	}
	spin_unlock_irqrestore(&alarm_slock, flags);
}
```

函数 alarm_timer_triggered()在文件 alarm.c 中定义。

（5）函数 alarm_ioctl()的功能是提供如下的 ioctl 命令。

☑ ANDROID_ALARM_CLEAR：功能是清除 Alarm，即 deactivate 这个 Alarm。
☑ ANDROID_ALARM_SET_OLD：功能是设置 Alarm 闹铃时间。
☑ ANDROID_ALARM_SET：功能是设置 Alarm 闹铃时间。
☑ ANDROID_ALARM_SET_AND_WAIT_OLD：功能是设置 Alarm 闹铃时间并等待这个 Alarm。
☑ ANDROID_ALARM_SET_AND_WAIT：功能是设置 Alarm 闹铃时间并等待这个 Alarm。
☑ ANDROID_ALARM_WAIT：功能是等待 Alarm。
☑ ANDROID_ALARM_SET_RTC：功能是设置 RTC 时间。
☑ ANDROID_ALARM_GET_TIME：功能是读取 Alarm 时间。

函数 alarm_ioctl()的具体实现代码如下。

```
static long alarm_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{

	struct timespec ts;
	int rv;

	switch (ANDROID_ALARM_BASE_CMD(cmd)) {
	case ANDROID_ALARM_SET_AND_WAIT(0):
	case ANDROID_ALARM_SET(0):
	case ANDROID_ALARM_SET_RTC:
		if (copy_from_user(&ts, (void __user *)arg, sizeof(ts)))
			return -EFAULT;
		break;
	}

	rv = alarm_do_ioctl(file, cmd, &ts);
	if (rv)
```

```
                return rv;

        switch (ANDROID_ALARM_BASE_CMD(cmd)) {
        case ANDROID_ALARM_GET_TIME(0):
                if (copy_to_user((void __user *)arg, &ts, sizeof(ts)))
                        return -EFAULT;
                break;
        }

        return 0;
}
```

（6）函数 alarm_do_ioctl()的功能是根据 switch…case 语句执行对应的 iotc 命令，具体实现代码如下。

```
static long alarm_do_ioctl(struct file *file, unsigned int cmd,
                                                        struct timespec *ts)
{
        int rv = 0;
        unsigned long flags;
        enum android_alarm_type alarm_type = ANDROID_ALARM_IOCTL_TO_TYPE(cmd);

        if (alarm_type >= ANDROID_ALARM_TYPE_COUNT)
                return -EINVAL;

        if (ANDROID_ALARM_BASE_CMD(cmd) != ANDROID_ALARM_GET_TIME(0)) {
                if ((file->f_flags & O_ACCMODE) == O_RDONLY)
                        return -EPERM;
                if (file->private_data == NULL &&
                    cmd != ANDROID_ALARM_SET_RTC) {
                        spin_lock_irqsave(&alarm_slock, flags);
                        if (alarm_opened) {
                                spin_unlock_irqrestore(&alarm_slock, flags);
                                return -EBUSY;
                        }
                        alarm_opened = 1;
                        file->private_data = (void *)1;
                        spin_unlock_irqrestore(&alarm_slock, flags);
                }
        }

        switch (ANDROID_ALARM_BASE_CMD(cmd)) {
        case ANDROID_ALARM_CLEAR(0):
                alarm_clear(alarm_type);
                break;
        case ANDROID_ALARM_SET(0):
                alarm_set(alarm_type, ts);
                break;
        case ANDROID_ALARM_SET_AND_WAIT(0):
                alarm_set(alarm_type, ts);
                /* fall though */
        case ANDROID_ALARM_WAIT:
                rv = alarm_wait();
                break;
```

```
        case ANDROID_ALARM_SET_RTC:
                rv = alarm_set_rtc(ts);
                break;
        case ANDROID_ALARM_GET_TIME(0):
                rv = alarm_get_time(alarm_type, ts);
                break;

        default:
                rv = -EINVAL;
        }
        return rv;
}
```

（7）函数 alarm_compat_ioctl()的功能是根据 switch…case 语句实现 iotc 指令的兼容性处理，具体实现代码如下。

```
static long alarm_compat_ioctl(struct file *file, unsigned int cmd,
                                                        unsigned long arg)
{

        struct timespec ts;
        int rv;

        switch (ANDROID_ALARM_BASE_CMD(cmd)) {
        case ANDROID_ALARM_SET_AND_WAIT_COMPAT(0):
        case ANDROID_ALARM_SET_COMPAT(0):
        case ANDROID_ALARM_SET_RTC_COMPAT:
                if (compat_get_timespec(&ts, (void __user *)arg))
                        return -EFAULT;
                /* fall through */
        case ANDROID_ALARM_GET_TIME_COMPAT(0):
                cmd = ANDROID_ALARM_COMPAT_TO_NORM(cmd);
                break;
        }

        rv = alarm_do_ioctl(file, cmd, &ts);
        if (rv)
                return rv;

        switch (ANDROID_ALARM_BASE_CMD(cmd)) {
        case ANDROID_ALARM_GET_TIME(0): /* NOTE: we modified cmd above */
                if (compat_put_timespec(&ts, (void __user *)arg))
                        return -EFAULT;
                break;
        }

        return 0;
}
```

（8）分析各个 iotc 指令对应的处理函数，具体实现代码如下。

```
static void devalarm_start(struct devalarm *alrm, ktime_t exp)
{
        if (is_wakeup(alrm->type))
```

```
			alarm_start(&alrm->u.alrm, exp);
		else
			hrtimer_start(&alrm->u.hrt, exp, HRTIMER_MODE_ABS);
}


static int devalarm_try_to_cancel(struct devalarm *alrm)
{
		if (is_wakeup(alrm->type))
			return alarm_try_to_cancel(&alrm->u.alrm);
		return hrtimer_try_to_cancel(&alrm->u.hrt);
}

static void devalarm_cancel(struct devalarm *alrm)
{
		if (is_wakeup(alrm->type))
			alarm_cancel(&alrm->u.alrm);
		else
			hrtimer_cancel(&alrm->u.hrt);
}

static void alarm_clear(enum android_alarm_type alarm_type)
{
		uint32_t alarm_type_mask = 1U << alarm_type;
		unsigned long flags;

		spin_lock_irqsave(&alarm_slock, flags);
		alarm_dbg(IO, "alarm %d clear\n", alarm_type);
		devalarm_try_to_cancel(&alarms[alarm_type]);
		if (alarm_pending) {
			alarm_pending &= ~alarm_type_mask;
			if (!alarm_pending && !wait_pending)
				__pm_relax(&alarm_wake_lock);
		}
		alarm_enabled &= ~alarm_type_mask;
		spin_unlock_irqrestore(&alarm_slock, flags);

}

static void alarm_set(enum android_alarm_type alarm_type,
							struct timespec *ts)
{
		uint32_t alarm_type_mask = 1U << alarm_type;
		unsigned long flags;

		spin_lock_irqsave(&alarm_slock, flags);
		alarm_dbg(IO, "alarm %d set %ld.%09ld\n",
				alarm_type, ts->tv_sec, ts->tv_nsec);
		alarm_enabled |= alarm_type_mask;
		devalarm_start(&alarms[alarm_type], timespec_to_ktime(*ts));
		spin_unlock_irqrestore(&alarm_slock, flags);
```

```
}

static int alarm_wait(void)
{
        unsigned long flags;
        int rv = 0;

        spin_lock_irqsave(&alarm_slock, flags);
        alarm_dbg(IO, "alarm wait\n");
        if (!alarm_pending && wait_pending) {
                __pm_relax(&alarm_wake_lock);
                wait_pending = 0;
        }
        spin_unlock_irqrestore(&alarm_slock, flags);

        rv = wait_event_interruptible(alarm_wait_queue, alarm_pending);
        if (rv)
                return rv;

        spin_lock_irqsave(&alarm_slock, flags);
        rv = alarm_pending;
        wait_pending = 1;
        alarm_pending = 0;
        spin_unlock_irqrestore(&alarm_slock, flags);

        return rv;
}
static int alarm_set_rtc(struct timespec *ts)
{
        struct rtc_time new_rtc_tm;
        struct rtc_device *rtc_dev;
        unsigned long flags;
        int rv = 0;

        rtc_time_to_tm(ts->tv_sec, &new_rtc_tm);
        rtc_dev = alarmtimer_get_rtcdev();
        rv = do_settimeofday(ts);
        if (rv < 0)
                return rv;
        if (rtc_dev)
                rv = rtc_set_time(rtc_dev, &new_rtc_tm);

        spin_lock_irqsave(&alarm_slock, flags);
        alarm_pending |= ANDROID_ALARM_TIME_CHANGE_MASK;
        wake_up(&alarm_wait_queue);
        spin_unlock_irqrestore(&alarm_slock, flags);

        return rv;
}

static int alarm_get_time(enum android_alarm_type alarm_type,
```

```
                                                    struct timespec *ts)
{
        int rv = 0;

        switch (alarm_type) {
        case ANDROID_ALARM_RTC_WAKEUP:
        case ANDROID_ALARM_RTC:
                getnstimeofday(ts);
                break;
        case ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP:
        case ANDROID_ALARM_ELAPSED_REALTIME:
                get_monotonic_boottime(ts);
                break;
        case ANDROID_ALARM_SYSTEMTIME:
                ktime_get_ts(ts);
                break;
        default:
                rv = -EINVAL;
        }
        return rv;
}
```

15.4　JNI 层详解

在 Alarm 系统中，本地 JNI 部分的实现文件是 frameworks/base/services/jni/com_android_server_AlarmManagerService.cpp，此文件是 Alarm 部分的本地代码，同时提供了 JNI 的接口。JNI 层的功能是从底层调用驱动程序，向上面的应用层提供 JNI 接口。文件 com_android_server_AlarmManagerService.cpp 的具体实现代码如下。

```
//设置内核时区接口
static jint android_server_AlarmManagerService_setKernelTimezone(JNIEnv* env, jobject obj, jint fd, jint minswest)
{
    struct timezone tz;
    tz.tz_minuteswest = minswest;
    tz.tz_dsttime = 0;
    int result = settimeofday(NULL, &tz);
    if (result < 0) {
        ALOGE("Unable to set kernel timezone to %d: %s\n", minswest, strerror(errno));
        return -1;
    } else {
        ALOGD("Kernel timezone updated to %d minutes west of GMT\n", minswest);
    }
    return 0;
}
//初始化接口
static jint android_server_AlarmManagerService_init(JNIEnv* env, jobject obj)
{
    return open("/dev/alarm", O_RDWR);
```

```
}
//关闭接口
static void android_server_AlarmManagerService_close(JNIEnv* env, jobject obj, jint fd)
{
        close(fd);
}

//设置闹钟接口
static void android_server_AlarmManagerService_set(JNIEnv* env, jobject obj, jint fd, jint type, jlong seconds,
jlong nanoseconds)
{
       struct timespec ts;
       ts.tv_sec = seconds;
       ts.tv_nsec = nanoseconds;
        int result = ioctl(fd, ANDROID_ALARM_SET(type), &ts);
        if (result < 0)
        {
            ALOGE("Unable to set alarm to %lld.%09lld: %s\n", seconds, nanoseconds, strerror(errno));
       }
}
//等待时钟接口
static jint android_server_AlarmManagerService_waitForAlarm(JNIEnv* env, jobject obj, jint fd)
{
        int result = 0;
        do
        {
                result = ioctl(fd, ANDROID_ALARM_WAIT);
        } while (result < 0 && errno == EINTR);

        if (result < 0)
        {
            ALOGE("Unable to wait on alarm: %s\n", strerror(errno));
            return 0;
       }

       return result;
}
//下面是定义接口的代码
static JNINativeMethod sMethods[] = {
         /* name, signature, funcPtr */
        {"init", "()I", (void*)android_server_AlarmManagerService_init},
        {"close", "(I)V", (void*)android_server_AlarmManagerService_close},
        {"set", "(IIJJ)V", (void*)android_server_AlarmManagerService_set},
       {"waitForAlarm", "(I)I", (void*)android_server_AlarmManagerService_waitForAlarm},
       {"setKernelTimezone", "(II)I", (void*)android_server_AlarmManagerService_setKernelTimezone},
};
//注册闹钟服务
int register_android_server_AlarmManagerService(JNIEnv* env)
{
       return jniRegisterNativeMethods(env, "com/android/server/AlarmManagerService",
                                                sMethods, NELEM(sMethods));
```

```
}
} /* namespace android */
```

15.5　Java 层详解

在 Android 系统中，Alarm 系统的 Java 层的实现文件如下。

☑　frameworks/base/services/java/com/android/server/AlarmManagerService.java

☑　frameworks/base/core/java/android/app/AlarmManager.java

本节将详细讲解上述文件的具体实现流程。

15.5.1　分析 AlarmManagerService 类

在 Android 系统中，闹钟和唤醒功能都是由 Alarm Manager Service 控制并管理的，RTC 闹钟以及定时器都和它有莫大的关系。为了方便，常常把这个 service 简称为 ALMS。ALMS 提供了一个 AlarmManager 辅助类。在实际代码中，应用程序一般都是通过这个辅助类来和 ALMS 打交道的。对具体的实现代码来说，辅助类只不过是把一些逻辑语义传递给 ALMS 服务端而已，具体怎么做则完全要看 ALMS 的实现代码了。

因为 ALMS 是服务端的内容，所以必须向外提供具体的接口才能被外界使用。在 Android 平台中，ALMS 的外部接口为 IAlarmManager，在 frameworks/base/core/java/android/app/IAlarmManager.aidl 脚本中定义。

具体的定义代码如下。

```
interface IAlarmManager {
    void set(int type, long triggerAtTime, in PendingIntent operation);
    void setRepeating(int type, long triggerAtTime, long interval, in PendingIntent operation);
    void setInexactRepeating(int type, long triggerAtTime, long interval, in PendingIntent operation);
    void setTime(long millis);
    void setTimeZone(String zone);
    void remove(in PendingIntent operation);
}
```

在大多数情况下，Service 的使用者会通过 Service Manager Service 接口，先获取感兴趣的 Service 对应的代理 I 接口，然后再调用 I 接口的成员函数向 Service 发出请求，所以应该先拿到一个 IAlarmManager 接口，然后再使用它。可是对于 Alarm Manager Service 来说，具体实现情况有所不同，最常见的调用方式如下。

```
manager = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
```

其中，函数 getSystemService()返回的不再是 IAlarmManager 接口，而是 AlarmManager 对象。

在文件 AlarmManagerService.java 中实现了 android.server 包中的 AlarmManagerService 类，具体实现流程如下。

（1）看类 Alarm，功能是定义了逻辑闹钟类，具体代码如下所示。

```
private static class Alarm {
    public int type;
    public int count;
    public long when;
    public long repeatInterval;
    public PendingIntent operation;
    public int uid;
    public int pid;
    ...
```

由此可见，在其中记录了逻辑闹钟的一些关键信息，具体说明如下。

☑ type 域：记录着逻辑闹钟的闹钟类型，例如 RTC_WAKEUP、ELAPSED_REALTIME_WAKEUP 等。
☑ count 域：是个辅助域，它和 repeatInterval 域一起工作。当 repeatInterval 大于 0 时，这个域可被用于计算下一次重复激发 Alarm 的时间。这是针对重复性闹钟的一个辅助域，重复性闹钟的实现机理是，如果当前时刻已经超过闹钟的激发时刻，那么 ALMS 会先从逻辑闹钟数组中摘取下 Alarm 节点，并执行闹钟对应的逻辑动作，然后进一步比较“当前时刻”和 Alarm 理应激发的“理想时刻”之间的时间跨度，从而计算出 Alarm 的“下一次理应激发的理想时刻”，并将这个激发时间记入 Alarm 节点，接着将该节点重新排入逻辑闹钟列表。这一点和普通 Alarm 不一样，普通 Alarm 节点摘下后就不再归还回逻辑闹钟列表。
☑ when 域：记录闹钟的激发时间，这个域和 type 域相关。
☑ repeatInterval 域：表示重复激发闹钟的时间间隔，如果闹钟只需激发一次，则此域为 0；如果闹钟需要重复激发，则此域为以毫秒为单位的时间间隔。
☑ operation 域：记录闹钟激发时应该执行的动作。
☑ uid 域：记录设置闹钟的进程的 uid。
☑ pid 域：记录设置闹钟的进程的 pid。

（2）函数 set()的功能是设置闹钟，能够设置一次性闹钟，具体实现代码如下。

```
public void set(int type, long triggerAtTime, PendingIntent operation) {
    setRepeating(type, triggerAtTime, 0, operation);
}
```

在上述代码中，第 1 个参数表示闹钟类型，第 2 个参数表示闹钟执行时间，第 3 个参数表示闹钟响应动作。

（3）函数 setRepeating()的功能是设置周期性的闹钟，具体实现代码如下。

```
public void setRepeating(int type, long triggerAtTime, long interval,
        PendingIntent operation) {
    if (operation == null) {
        Slog.w(TAG, "set/setRepeating ignored because there is no intent");
        return;
    }
    synchronized (mLock) {
        Alarm alarm = new Alarm();
        alarm.type = type;
        alarm.when = triggerAtTime;
        alarm.repeatInterval = interval;
        alarm.operation = operation;

        // Remove this alarm if already scheduled.
        removeLocked(operation);

        if (localLOGV) Slog.v(TAG, "set: " + alarm);

        int index = addAlarmLocked(alarm);
        if (index == 0) {
            setLocked(alarm);
        }
    }
}
```

通过上述代码可知，函数 setRepeating()可以设置重复闹钟，其中第 1 个参数表示闹钟类型，第 2 个参数表示闹钟首次执行时间，第 3 个参数表示闹钟两次执行的间隔时间，第 4 个参数表示闹钟响应动作。其实现原理类似 Java 的 Timer 中的 scheduleAtFixedRate(TimerTask task, long delay, long period)，都是以近似固定的时间间隔（由指定的周期分隔）进行后续执行。在固定速率执行中，根据已安排的初始执行时间来安排每次执行。如果由于任何原因（如垃圾回收或其他后台活动）而延迟了某次执行，则将快速连续地出现两次或更多的执行，从而使后续执行能够“追上来”。

（4）函数 setInexactRepeating()的功能也是设置重复闹钟，与函数 setRepeating()的功能相似，不过其两个闹钟执行的间隔时间不是固定的而已。函数 setInexactRepeating()相对而言更节能（power-efficient）一些，因为系统可能会将几个差不多的闹钟合并为一个来执行，减少设备的唤醒次数。这一点类似 Java 的 Timer 中的 schedule(TimerTask task, Date firstTime, long period)，能够根据前一次执行的实际执行时间来安排每次执行。如果由于任何原因（如垃圾回收或其他后台活动）而延迟了某次执行，则后续执行也将被延迟。在长期运行中，执行的频率一般要稍慢于指定周期的倒数（假定 Object.wait(long)所依靠的系统时钟是准确的）。函数 setInexactRepeating()的具体实现代码如下。

```
public void setInexactRepeating(int type, long triggerAtTime, long interval,
            PendingIntent operation) {
        if (operation == null) {
            Slog.w(TAG, "setInexactRepeating ignored because there is no intent");
            return;
        }

        if (interval <= 0) {
            Slog.w(TAG, "setInexactRepeating ignored because interval " + interval
                    + " is invalid");
            return;
        }

        // If the requested interval isn't a multiple of 15 minutes, just treat it as exact
        if (interval % QUANTUM != 0) {
            if (localLOGV) Slog.v(TAG, "Interval " + interval + " not a quantum multiple");
            setRepeating(type, triggerAtTime, interval, operation);
            return;
        }

        // Translate times into the ELAPSED timebase for alignment purposes so that
        // alignment never tries to match against wall clock times.
        final boolean isRtc = (type == AlarmManager.RTC || type == AlarmManager.RTC_WAKEUP);
        final long skew = (isRtc)
                ? System.currentTimeMillis() - SystemClock.elapsedRealtime()
                : 0;

        // Slip forward to the next ELAPSED-timebase quantum after the stated time.   If
        // we're *at* a quantum point, leave it alone.
        final long adjustedTriggerTime;
        long offset = (triggerAtTime - skew) % QUANTUM;
        if (offset != 0) {
            adjustedTriggerTime = triggerAtTime - offset + QUANTUM;
        } else {
            adjustedTriggerTime = triggerAtTime;
```

```
        }

        // Set the alarm based on the quantum-aligned start time
        if (localLOGV) Slog.v(TAG, "setInexactRepeating: type=" + type + " interval=" + interval
                + " trigger=" + adjustedTriggerTime + " orig=" + triggerAtTime);
        setRepeating(type, adjustedTriggerTime, interval, operation);
    }
```

在上述代码中，参数 int type 表示闹钟的类型，常用的类型有如下 5 个值。

☑ AlarmManager.ELAPSED_REALTIME：表示当系统进入睡眠状态时，这种类型的闹铃不会唤醒系统。直到系统下次被唤醒才传递它，该闹铃所用的时间是相对时间，是从系统启动后开始计时的，包括睡眠时间，可以通过调用 SystemClock.elapsedRealtime()获得。系统值是 3(0x00000003)。

☑ AlarmManager.ELAPSED_REALTIME_WAKEUP：表示闹钟在睡眠状态下会唤醒系统并执行提示功能，该状态下闹钟也使用相对时间，用法同 ELAPSED_REALTIME，系统值是 2(0x00000002)。

☑ AlarmManager.RTC：表示闹钟在睡眠状态下，这种类型的闹铃不会唤醒系统。直到系统下次被唤醒才传递它，该闹铃所用的时间是绝对时间，所用时间是UTC时间，可以通过调用 System.currentTimeMillis()获得。系统值是 1(0x00000001)。

☑ AlarmManager.RTC_WAKEUP：表示闹钟在睡眠状态下会唤醒系统并执行提示功能，该状态下闹钟使用绝对时间，系统值为 0(0x00000000)。

☑ AlarmManager.POWER_OFF_WAKEUP：表示闹钟在手机关机状态下也能正常进行提示功能（关机闹钟），所以是 5 个状态中用的最多的状态之一，该状态下闹钟也是用绝对时间，系统值为 4(0x00000004)；不过本状态好像受 SDK 版本影响，注意某些版本并不支持此类型。

（5）函数 setTime()的功能是设置时间，具体实现代码如下。

```
public void setTime(long millis) {
    mContext.enforceCallingOrSelfPermission(
            "android.permission.SET_TIME",
            "setTime");

    SystemClock.setCurrentTimeMillis(millis);
}
```

（6）函数 setTimeZone()的功能是设置时区，此功能需要 android.permission.SET_TIME_ZONE 权限，具体实现代码如下。

```
public void setTimeZone(String tz) {
    mContext.enforceCallingOrSelfPermission(
            "android.permission.SET_TIME_ZONE",
            "setTimeZone");

    long oldId = Binder.clearCallingIdentity();
    try {
        if (TextUtils.isEmpty(tz)) return;
        TimeZone zone = TimeZone.getTimeZone(tz);
        // Prevent reentrant calls from stepping on each other when writing
        // the time zone property
        boolean timeZoneWasChanged = false;
        synchronized (this) {
            String current = SystemProperties.get(TIMEZONE_PROPERTY);
            if (current == null || !current.equals(zone.getID())) {
                if (localLOGV) {
```

```
                Slog.v(TAG, "timezone changed: " + current + ", new=" + zone.getID());
            }
            timeZoneWasChanged = true;
            SystemProperties.set(TIMEZONE_PROPERTY, zone.getID());
        }
        // Update the kernel timezone information
        // Kernel tracks time offsets as 'minutes west of GMT'
        int gmtOffset = zone.getOffset(System.currentTimeMillis());
        setKernelTimezone(mDescriptor, -(gmtOffset / 60000));
        }

    TimeZone.setDefault(null);

    if (timeZoneWasChanged) {
        Intent intent = new Intent(Intent.ACTION_TIMEZONE_CHANGED);
        intent.addFlags(Intent.FLAG_RECEIVER_REPLACE_PENDING);
        intent.putExtra("time-zone", zone.getID());
        mContext.sendBroadcastAsUser(intent, UserHandle.ALL);
    }
  } finally {
    Binder.restoreCallingIdentity(oldId);
  }
}
```

（7）函数 remove()的功能是取消某一个闹钟，在取消 Alarm 时，是以一个 PendingIntent 对象作为参数进行传递的。这个 PendingIntent 对象正是当初设置 Alarm 时所传入 operation 参数。不能随便创建一个新的 PendingIntent 对象来调用函数 remove()，否则 remove 不会起任何作用。对象 PendingIntent 在 AMS（Activity Manager Service）端对应一个 PendingIntentRecord 实体，而 ALMS 在遍历逻辑闹钟列表时，会根据是否指代相同的 PendingIntentRecord 实体来判断 PendingIntent 是否相符。如果随便地创建一个 PendingIntent 对象并传入函数 remove()，那么在 ALMS 端会找不到相符的 PendingIntent 对象，所以 remove 肯定不会起任何作用。函数 remove()的具体实现代码如下。

```
public void removeLocked(PendingIntent operation) {
    removeLocked(mRtcWakeupAlarms, operation);
    removeLocked(mRtcAlarms, operation);
    removeLocked(mElapsedRealtimeWakeupAlarms, operation);
    removeLocked(mElapsedRealtimeAlarms, operation);
}
```

在上述代码中，把 4 个逻辑闹钟数组都遍历一遍，删除其中所有和 operation 相符的 Alarm 节点。

（8）函数 removeLocked()的功能是取消闹钟列表中的某个闹钟，具体实现代码如下。

```
private void removeLocked(ArrayList<Alarm> alarmList,
        PendingIntent operation) {
    if (alarmList.size() <= 0) {
        return;
    }

    // iterator over the list removing any it where the intent match
    Iterator<Alarm> it = alarmList.iterator();

    while (it.hasNext()) {
        Alarm alarm = it.next();
        if (alarm.operation.equals(operation)) {
```

```
                it.remove();
            }
        }
    }
```

（9）函数 removeUserLocked()的功能是取消用户设置的闹钟，具体实现代码如下。

```
private void removeUserLocked(ArrayList<Alarm> alarmList, int userHandle) {
    if (alarmList.size() <= 0) {
        return;
    }

    // iterator over the list removing any it where the intent match
    Iterator<Alarm> it = alarmList.iterator();

    while (it.hasNext()) {
        Alarm alarm = it.next();
        if (UserHandle.getUserId(alarm.operation.getCreatorUid()) == userHandle) {
            it.remove();
        }
    }
}
```

通过前面的取消闹钟函数的实现代码可以看出，所谓的取消工作只是删除了对应的逻辑 Alarm 节点而已，并不会和底层驱动再打什么交道。也就是说，是不存在针对底层“实体闹钟”的删除动作的。所以在底层“实体闹钟”到时还是会被“激发”出来，只不过此时在 Frameworks 层会因为找不到符合要求的“逻辑闹钟”而不做进一步的激发动作而已。

（10）函数 addAlarmLocked()的功能是将逻辑闹钟添加到内部逻辑闹钟数组的某个合适位置，具体实现代码如下。

```
private int addAlarmLocked(Alarm alarm) {
    ArrayList<Alarm> alarmList = getAlarmList(alarm.type);

    int index = Collections.binarySearch(alarmList, alarm, mIncreasingTimeOrder);
    if (index < 0) {
        index = 0 - index - 1;
    }
    if (localLOGV) Slog.v(TAG, "Adding alarm " + alarm + " at " + index);
    alarmList.add(index, alarm);

    if (localLOGV) {
        // Display the list of alarms for this alarm type
        Slog.v(TAG, "alarms: " + alarmList.size() + " type: " + alarm.type);
        int position = 0;
        for (Alarm a : alarmList) {
            Time time = new Time();
            time.set(a.when);
            String timeStr = time.format("%b %d %I:%M:%S %p");
            Slog.v(TAG, position + ": " + timeStr
                    + " " + a.operation.getTargetPackage());
            position += 1;
        }
    }
```

```
    return index;
}
```

在 Android 系统中，逻辑闹钟列表是依据 Alarm 的激发时间实现排序的。因为越早被激发的 Alarm 越靠近第 0 位。所以函数 addAlarmLocked()在添加新逻辑闹钟时，需要先用二分查找法快速找到列表中合适的位置，然后再把 Alarm 对象插入此处。

（11）类 AlarmThread 是在 AlarmManagerService 中的一个继承于 Thread 的内嵌类，具体定义代码如下。

```
private class AlarmThread extends Thread
```

类 AlarmThread 的核心是函数 run()，具体实现流程如下所示。

☑ 在 AlarmThread 线程中，在一个 while(true)循环中不断调用函数 waitForAlarm()来等待底层 Alarm 激发动作。当 AlarmThread 调用到函数 ioctl()时线程会被阻塞住，直到底层激发 Alarm。而且所激发的 Alarm 的类型会记录到 ioctl()的返回值中。这个返回值对外界来说非常重要，外界用它来判断该遍历哪个逻辑闹钟列表。

☑ 一旦等到底层驱动的激发动作，AlarmThread 会开始遍历相应的逻辑闹钟列表。AlarmThread 先创建了一个临时的数组列表 triggerList，然后根据 result 的值对相应的 Alarm 数组列表调用函数 triggerAlarmsLocked()。如果发现 Alarm 数组列表中某个 Alarm 符合激发条件，则把它移到 triggerList 中。这样，4 条 Alarm 数组列表中需要激发的 Alarm 就汇总到 triggerList 数组列表中。

☑ 遍历一遍 triggerList，每当在 while 循环中遍历到一个 Alarm 对象，就执行它的 alarm.operation.send()函数。在 Alarm 中记录的 operation，就是当初设置它时传来的那个 PendingIntent 对象。

函数 run()的具体实现代码如下。

```
public void run()
        {
            while (true)
            {
                int result = waitForAlarm(mDescriptor);

                ArrayList<Alarm> triggerList = new ArrayList<Alarm>();

                if ((result & TIME_CHANGED_MASK) != 0) {
                    remove(mTimeTickSender);
                    mClockReceiver.scheduleTimeTickEvent();
                    Intent intent = new Intent(Intent.ACTION_TIME_CHANGED);
                    intent.addFlags(Intent.FLAG_RECEIVER_REPLACE_PENDING
                            | Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
                    mContext.sendBroadcastAsUser(intent, UserHandle.ALL);
                }

                synchronized (mLock) {
                    final long nowRTC = System.currentTimeMillis();
                    final long nowELAPSED = SystemClock.elapsedRealtime();
                    if (localLOGV) Slog.v(
                        TAG, "Checking for alarms... rtc=" + nowRTC
                        + ", elapsed=" + nowELAPSED);

                    if ((result & RTC_WAKEUP_MASK) != 0)
                        triggerAlarmsLocked(mRtcWakeupAlarms, triggerList, nowRTC);

                    if ((result & RTC_MASK) != 0)
                        triggerAlarmsLocked(mRtcAlarms, triggerList, nowRTC);
```

```
        if ((result & ELAPSED_REALTIME_WAKEUP_MASK) != 0)
            triggerAlarmsLocked(mElapsedRealtimeWakeupAlarms, triggerList, nowELAPSED);

        if ((result & ELAPSED_REALTIME_MASK) != 0)
            triggerAlarmsLocked(mElapsedRealtimeAlarms, triggerList, nowELAPSED);

        // now trigger the alarms
        Iterator<Alarm> it = triggerList.iterator();
        while (it.hasNext()) {
            Alarm alarm = it.next();
            try {
                if (localLOGV) Slog.v(TAG, "sending alarm " + alarm);
                alarm.operation.send(mContext, 0,
                        mBackgroundIntent.putExtra(
                                Intent.EXTRA_ALARM_COUNT, alarm.count),
                        mResultReceiver, mHandler);

                // we have an active broadcast so stay awake.
                if (mBroadcastRefCount == 0) {
                    setWakelockWorkSource(alarm.operation);
                    mWakeLock.acquire();
                }
                final InFlight inflight = new InFlight(AlarmManagerService.this,
                        alarm.operation);
                mInFlight.add(inflight);
                mBroadcastRefCount++;

                final BroadcastStats bs = inflight.mBroadcastStats;
                bs.count++;
                if (bs.nesting == 0) {
                    bs.nesting = 1;
                    bs.startTime = nowELAPSED;
                } else {
                    bs.nesting++;
                }
                final FilterStats fs = inflight.mFilterStats;
                fs.count++;
                if (fs.nesting == 0) {
                    fs.nesting = 1;
                    fs.startTime = nowELAPSED;
                } else {
                    fs.nesting++;
                }
                if (alarm.type == AlarmManager.ELAPSED_REALTIME_WAKEUP
                        || alarm.type == AlarmManager.RTC_WAKEUP) {
                    bs.numWakeup++;
                    fs.numWakeup++;
                    ActivityManagerNative.noteWakeupAlarm(
                            alarm.operation);
                }
```

```
                    } catch (PendingIntent.CanceledException e) {
                        if (alarm.repeatInterval > 0) {
                            // This IntentSender is no longer valid, but this
                            // is a repeating alarm, so toss the hoser.
                            remove(alarm.operation);
                        }
                    } catch (RuntimeException e) {
                        Slog.w(TAG, "Failure sending alarm.", e);
                    }
                }
            }
        }
    }
}
```

在上述代码中调用了函数 noteWakeupAlarm()，其功能是调用 BatteryStatsService 服务的相关动作并导致机器的唤醒。函数 noteWakeupAlarm()的具体实现代码如下。

```
public void noteWakeupAlarm(IIntentSender sender)
{
    if (!(sender instanceof PendingIntentRecord))
    {
        return;
    }

    BatteryStatsImpl stats = mBatteryStatsService.getActiveStatistics();
    synchronized (stats)
    {
        if (mBatteryStatsService.isOnBattery())
        {
            mBatteryStatsService.enforceCallingPermission();
            PendingIntentRecord rec = (PendingIntentRecord)sender;
            int MY_UID = Binder.getCallingUid();
            int uid = rec.uid == MY_UID ? Process.SYSTEM_UID : rec.uid;
            BatteryStatsImpl.Uid.Pkg pkg = stats.getPackageStatsLocked(uid, rec.key.packageName);
            pkg.incWakeupsLocked();
        }
    }
}
```

15.5.2 分析 AlarmManager 类

在文件 AlarmManager.java 中，通过使用 AlarmManagerService 服务实现了 android.app 包中的 AlarmManager 类，并对 Java 层提供了平台 API。在 AlarmManager 中提供了一个构造函数、6 个功能函数，这基本上和 IAlarmManager 的成员函数一一对应。

```
AlarmManager(IAlarmManager service)
public void set(int type, long triggerAtTime, PendingIntent operation)
public void setRepeating(int type, long triggerAtTime, long interval,
PendingIntent operation)
public void setInexactRepeating(int type, long triggerAtTime, long interval,
                                        PendingIntent operation)
public void cancel(PendingIntent operation)
```

```
public void setTime(long millis)
public void setTimeZone(String timeZone)
```

下面将详细讲解文件 AlarmManager.java 的具体实现流程。

（1）定义常量和接口，具体实现代码如下。

```
public static final int RTC_WAKEUP = 0;
public static final int RTC = 1;
public static final int ELAPSED_REALTIME_WAKEUP = 2;
public static final int ELAPSED_REALTIME = 3;
private final IAlarmManager mService;
```

（2）在文件 AlarmManager.java 中提供了闹钟操作的各个接口函数，这个接口函数是通过调用对应在文件 AlarmManager.java 中的函数实现的。各个接口函数的具体实现代码如下。

```
    public void set(int type, long triggerAtMillis, PendingIntent operation) {
        try {
            mService.set(type, triggerAtMillis, operation);
        } catch (RemoteException ex) {
        }
    }
    public void setRepeating(int type, long triggerAtMillis,
            long intervalMillis, PendingIntent operation) {
        try {
            mService.setRepeating(type, triggerAtMillis, intervalMillis, operation);
        } catch (RemoteException ex) {
        }
    }
    public void setInexactRepeating(int type, long triggerAtMillis,
            long intervalMillis, PendingIntent operation) {
        try {
            mService.setInexactRepeating(type, triggerAtMillis, intervalMillis, operation);
        } catch (RemoteException ex) {
        }
    }
    public void cancel(PendingIntent operation) {
        try {
            mService.remove(operation);
        } catch (RemoteException ex) {
        }
    }
    public void setTime(long millis) {
        try {
            mService.setTime(millis);
        } catch (RemoteException ex) {
        }
    }
    public void setTimeZone(String timeZone) {
        try {
            mService.setTimeZone(timeZone);
        } catch (RemoteException ex) {
        }
    }
}
```

因为在本章前面讲解 AlarmManagerService.java 文件时已经讲解了各个操作函数的具体实现，在此不再讲解 AlarmManager.java 文件中的接口函数，因为它们的功能是一样的。

15.6　模拟器环境的具体实现

Alarm 系统和其他系统相比，比较特殊的是只有模拟器的 RTC 驱动程序，具体是在文件 drivers/rtc/rtc-goldfish.c 中实现的。Goldfish 的 Alarm 驱动由模拟器的虚拟环境触发中断，并填充相关的寄存器，在驱动程序中取得信息。

函数 goldfish_rtc_read_time()用于获取当前的时间，具体代码如下。

```
static int goldfish_rtc_read_time(struct device *dev, struct rtc_time *tm) {
int64_t time;
struct goldfish_rtc *qrtc = platform_get_drvdata(to_platform_device(dev));
time = readl(qrtc >base + TIMER_TIME_LOW);              /*读取虚拟的寄存器*/
time |= (int64_t)readl(qrtc >base + TIMER_TIME_HIGH) << 32;
do_div(time, NSEC_PER_SEC);
rtc_time_to_tm(time, tm);
return 0;
 }
```

在上述代码中，通过读取 TIME_TIME_LOW 和 TIME_TIME_HIGH 这两个虚拟寄存器来获得当前的时间。

在 MSM 平台中，Alarm 驱动程序是在文件 drivers/rtc/rtc-MSM7k00a.c 中实现的，具体的功能是调用 RPC（远程过程调用）完成的。通过函数 msmrtc_probe()实现探测初始化，具体代码如下所示。

```
static int msmrtc_probe(struct platform_device *pdev) {
 struct rpcsvr_platform_device *rdev =
container_of(pdev, struct rpcsvr_platform_device, base);
ep = msm_rpc_connect(rdev >prog, rdev >vers, 0);              /*连接到 RPC*/
...
rtc = rtc_device_register("msm_rtc", &pdev >dev,              /*建立 RTC 设备*/
          &msm_rtc_ops, THIS_MODULE);                          /* ......   省略部分错误处理的内容*/
 return 0;
}
```

msm_rtc_ops 是一个 rtc_class_ops 类型的结构体，在里面实现了成员 read_time、set_time 和 set_ alarm，其中函数 msmrtc_timeremote_read_time()负责读取当前的时间，具体代码如下。

```
static int msmrtc_timeremote_read_time(struct device *dev, struct rtc_time *tm){
        int rc;
struct timeremote_get_julian_req {     /*表示 RPC 请求的结构体*/
struct rpc_request_hdr hdr;
uint32_t julian_time_not_null;
} req;
struct timeremote_get_julian_rep {     /*表示 RPC 回应的结构体*/
struct rpc_reply_hdr hdr;
uint32_t opt_arg;
struct rpc_time_julian time;
} rep;
req.julian_time_not_null = cpu_to_be32(1);
rc = msm_rpc_call_reply(ep, TIMEREMOTE_PROCEEDURE_GET_JULIAN,   /*RPC 调用*/
&req,sizeof(req),&rep, sizeof(rep), 5 * HZ);
```

```
...
tm >tm_year = be32_to_cpu(rep.time.year);                    /*填充 rtc_time 结构*/
tm >tm_mon = be32_to_cpu(rep.time.month);
tm >tm_mday = be32_to_cpu(rep.time.day);
tm >tm_hour = be32_to_cpu(rep.time.hour);
tm >tm_min = be32_to_cpu(rep.time.minute);
tm >tm_sec = be32_to_cpu(rep.time.second);
tm >tm_wday = be32_to_cpu(rep.time.day_of_week);
tm >tm_year    = 1900;                                        /*设置从 1900 开始的 RTC*/
tm >tm_mon;                                                   /*RTC 月份的调整*/
...
return 0;
 }
```

具体功能的实现过程是通过统一的 RPC 在远端完成的，此处调用的是 RPC 的 TIMEREMOTE_PROCEEDURE_GET_JULIAN 命令。

15.7 实战演练

经过本章前面内容的学习，已经了解了 Android 系统中的警报器系统驱动 Alarm 的基本架构知识。本节将通过具体实例讲解开发和移植 Alarm 驱动的方法。

15.7.1 编写 PCF8563 芯片的 RTC 驱动程序

PCF8563 是 PHILIPS 公司推出的一款工业级内含 I2C 总线接口功能的具有极低功耗的多功能时钟/日历芯片。PCF8563 的多种报警功能、定时器功能、时钟输出功能以及中断输出功能能完成各种复杂的定时服务，甚至可为单片机提供看门狗功能，是一款性价比极高的时钟芯片，已被广泛用于电表、水表、气表、电话、传真机、便携式仪器以及电池供电的仪器仪表等产品领域。

因为在 PCF8563 中没有提供 RTC 驱动程序，所以需要个人编写驱动程序。具体实现流程如下。

（1）在 PCF8563 中提供了驱动文件 rtc_pcf8563.c，我们可以在此文件的基础上进行改写，其中文件 pcf8563.h 的具体实现代码如下。

```
#ifndef _PCF8563_H
#define _PCF8563_H
#define RTC_RD_TIME 0
#define RTC_SET_TIME 1
typedef struct
{
     int tm_sec;
     int tm_min;
     int tm_hour;
     int tm_mday;
     int tm_wday;
     int tm_mon;
     int tm_year;
}rtc_time_t;
#endif
```

（2）在文件 pcf8563.c 中定义函数 pcf8563_get_datetime()、pcf8563_set_datetime()和一个控制调用函数



pcf8563_rtc_ioctl()，主要实现代码如下。

```
//下面是宏定义
    #define PCF8563_REG_ST1 0x00 /* status */
    #define PCF8563_REG_ST2 0x01
    #define PCF8563_REG_SC 0x02 /* datetime */
    #define PCF8563_REG_MN 0x03
    #define PCF8563_REG_HR 0x04
    #define PCF8563_REG_DM 0x05
    #define PCF8563_REG_DW 0x06
    #define PCF8563_REG_MO 0x07
    #define PCF8563_REG_YR 0x08

    #define PCF8563_REG_AMN 0x09 /* alarm */
    #define PCF8563_REG_AHR 0x0A
    #define PCF8563_REG_ADM 0x0B
    #define PCF8563_REG_ADW 0x0C
    #define PCF8563_REG_CLKO 0x0D /* clock out */
    #define PCF8563_REG_TMRC 0x0E /* timer control */
    #define PCF8563_REG_TMR 0x0F /* timer */
    #define PCF8563_SC_LV 0x80 /* low voltage */
    #define PCF8563_MO_C 0x80 /* century */
    #define I2C_PCF8563 0xA2
static inline
int bin2bcd (int x)
{
return (x%10) | ((x/10) << 4);
}
static inline
int bcd2bin (int x)
{
return (x >> 4) * 10 + (x & 0x0f);
}
//下面的函数用于读 RTC pcf8563 数据
    static int pcf8563_get_datetime(int i2c_devaddress , rtc_time_t *tm)
    {
            unsigned char buf[PCF8563_REG_YR+1] = { PCF8563_REG_ST1 };
            /* status */
            //buf[PCF8563_REG_ST1] = hi_i2c_read(i2c_devaddress, PCF8563_REG_ST1);
            //buf[PCF8563_REG_ST2] = hi_i2c_read(i2c_devaddress, PCF8563_REG_ST2);

            /* datetime */
            buf[PCF8563_REG_SC] = hi_i2c_read(i2c_devaddress, PCF8563_REG_SC);
            buf[PCF8563_REG_MN] = hi_i2c_read(i2c_devaddress, PCF8563_REG_MN);
            buf[PCF8563_REG_HR] = hi_i2c_read(i2c_devaddress, PCF8563_REG_HR);
            buf[PCF8563_REG_DM] = hi_i2c_read(i2c_devaddress, PCF8563_REG_DM);
            buf[PCF8563_REG_DW] = hi_i2c_read(i2c_devaddress, PCF8563_REG_DW);
            buf[PCF8563_REG_MO] = hi_i2c_read(i2c_devaddress, PCF8563_REG_MO);
            buf[PCF8563_REG_YR] = hi_i2c_read(i2c_devaddress, PCF8563_REG_YR);
            buf[PCF8563_REG_ST1] = hi_i2c_read(i2c_devaddress, PCF8563_REG_ST1);

            tm->tm_sec = bcd2bin(buf[PCF8563_REG_SC] & 0x7F);
```

```
        tm->tm_min = bcd2bin(buf[PCF8563_REG_MN] & 0x7F);
        tm->tm_hour = bcd2bin(buf[PCF8563_REG_HR] & 0x3F); /* rtc hr 0-23 */
        tm->tm_mday = bcd2bin(buf[PCF8563_REG_DM] & 0x3F);
        tm->tm_wday = buf[PCF8563_REG_DW] & 0x07;
        tm->tm_mon = bcd2bin(buf[PCF8563_REG_MO] & 0x1F) - 1; /* rtc mn 1-12 */
        tm->tm_year = bcd2bin(buf[PCF8563_REG_YR]);

    #if 0
        printk("%s: secs=%d, mins=%d, hours=%d, "
                "mday=%d, mon=%d, year=%d, wday=%d\n",
                __func__,
                tm->tm_sec, tm->tm_min, tm->tm_hour,
                tm->tm_mday, tm->tm_mon, tm->tm_year, tm->tm_wday);
    #endif

        // Century bit. C = 0; indicates the century is 20xx
    // C = 1; indicates the century is 19xx
        if(buf[PCF8563_REG_MO] & PCF8563_MO_C)
            tm->tm_year += 1900;
        else
            tm->tm_year += 2000;
        return 0;
    }
//下面函数用于写 RTC pcf8563 数据
    static int pcf8563_set_datetime(int i2c_devaddress , rtc_time_t tm)
    {
        unsigned char buf[9];
        int Centurybit;
    #if 0
        printk("%s: secs=%d, mins=%d, hours=%d, "
                "mday=%d, mon=%d, year=%d, wday=%d\n",
                __func__,
                tm.tm_sec, tm.tm_min, tm.tm_hour,
                tm.tm_mday, tm.tm_mon, tm.tm_year, tm.tm_wday);
    #endif
        /* hours, minutes and seconds */
        buf[PCF8563_REG_SC] = bin2bcd(tm.tm_sec);
        buf[PCF8563_REG_MN] = bin2bcd(tm.tm_min);
        buf[PCF8563_REG_HR] = bin2bcd(tm.tm_hour);
        buf[PCF8563_REG_DM] = bin2bcd(tm.tm_mday);
        /* month, 1 - 12 */
        buf[PCF8563_REG_MO] = bin2bcd(tm.tm_mon + 1);
        /* year and century */
        buf[PCF8563_REG_YR] = bin2bcd(tm.tm_year % 100);

        if (tm.tm_year >= 1900 && tm.tm_year < 2000)
            Centurybit = PCF8563_MO_C;
        else if (tm.tm_year >= 2000 && tm.tm_year < 3000)
            Centurybit = 0;
        else
            return -1;
```

```
		// Century bit. C = 0; indicates the century is 20xx
		// C = 1; indicates the century is 19xx
		buf[PCF8563_REG_MO] |= Centurybit; //add Century bit
		buf[PCF8563_REG_DW] = tm.tm_wday & 0x07;
		/* write register's data */
		hi_i2c_write(i2c_devaddress, PCF8563_REG_SC, buf[PCF8563_REG_SC]);
		hi_i2c_write(i2c_devaddress, PCF8563_REG_MN, buf[PCF8563_REG_MN]);
		hi_i2c_write(i2c_devaddress, PCF8563_REG_HR, buf[PCF8563_REG_HR]);
		hi_i2c_write(i2c_devaddress, PCF8563_REG_DM, buf[PCF8563_REG_DM]);
		hi_i2c_write(i2c_devaddress, PCF8563_REG_MO, buf[PCF8563_REG_MO]);
		hi_i2c_write(i2c_devaddress, PCF8563_REG_YR, buf[PCF8563_REG_YR]);
		hi_i2c_write(i2c_devaddress, PCF8563_REG_DW, buf[PCF8563_REG_DW]);

		return 0;
	}
//下面是相关控制调用函数
	static int pcf8563_rtc_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
	{
	rtc_time_t tm;
	switch (cmd)
	{
	case RTC_RD_TIME:
	pcf8563_get_datetime(i2c_pcf8563, &tm);
	return copy_to_user((void *)arg, &tm, sizeof(tm)) ? -EFAULT : 0;
	case RTC_SET_TIME:
	if (copy_from_user(&tm, (struct rtc_time_t *) arg, sizeof(tm)))
	return -EFAULT;
	return pcf8563_set_datetime(i2c_pcf8563, tm);
	}
	return 0;
	}
	static struct file_operations pcf8563_fops = {
	.owner = THIS_MODULE,
	.ioctl = pcf8563_rtc_ioctl,
	.open = pcf8563_open,
	.release = pcf8563_close
	};
	static struct miscdevice pcf8563_dev = {
	MISC_DYNAMIC_MINOR,
	"pcf8563",
	&pcf8563_fops,
	};
```

15.7.2　在 2440 移植 RTC 驱动程序

接下来将以 2440 开发板为基础，讲解移植 RTC 驱动程序的具体方法。

（1）在文件 arch/arm/mach-s3c2440/mach-smdk2440.c 中添加 RTC 设备，在结构体 plat_device 中加入如下所示的代码。

```
&s3c_device_rtc,
```

（2）配置支持 RTC 工作的内核，具体配置信息如下所示。

```
Device Drivers --->
<*>Real Time Clock --->
[*]Set system time from RTC on startup and resume
(rtc0) rtc used to set the system time
[*]/sys/class/rtc/rtcN(sysfs)
[*]/proc/driver/rtc(procfs for rtc0)
[*]/dev/rtcN(character drivers)
<*>Samsung S3C series SoC RTC
```

在启动时会输出：

```
S3C24XX RTC, (c) 2004,2006 Simtec Electronics
s3c2410-rtc s3c2410-rtc: rtc disabled, re-enabling
s3c2410-rtc s3c2410-rtc: rtc core: registered s3c as rtc0
```

（3）在终端设备中用 busybox 自带的 date 命令来查看和设置时间，输出信息如下。

```
#date <—输入命令
Thu Jan 1 12:01:36 UTC 1970 <—显示的时间
#date -s 2014.10.22-16:30:10 <—设置时间格式：年.月.日-时:分:秒
Thu Oct 22 16:30:10 UTC 2014
#hwclock -w <—保存时间
```

在文件系统的启动脚本中加入如下所示的命令。

```
hwclock -s
```

这样即成功实现了系统移植，在每次启动系统时会自动同步硬件中的 RTC 时间。

```
s3c2410-rtc s3c2410-rtc: setting system clock to 2014-10-22 16:32:07 UTC
```

15.7.3 在 mini2440 开发板上的移植

RTC 时钟在 S3C2440 上的移植非常简单，因为 Linux 已经支持，仍以 platform 的形式来实现，只要把 RTC 的 platform_deivce 进行注册，并对内核进行简单配置即可。接下来将以 mini2440 开发板为基础，讲解移植 RTC 驱动程序的具体方法。

（1）在初始化文件中加入 RTC 设备结构

虽然 Linux-2.6.32.2 内核对 2440 的 RTC 驱动的支持已经十分完善了，但是并未在文件 mach- mini2440.c 中的设备集中加入它，所以并没有被激活，需要在 RTC 结构体中加入下面的粗斜体代码。

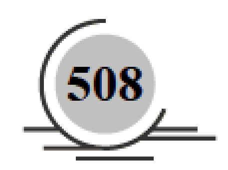
```
static struct platform_device *mini2440_devices[] __initdata = {
		&s3c_device_usb,
		&s3c_device_rtc,
		&s3c_device_lcd,
		&s3c_device_wdt,
		&s3c_device_i2c0,
		&s3c_device_iis,
		&mini2440_device_eth,
		&s3c_device_nand,
};
```

（2）在内核中配置 RTC

接下来重新配置内核，以加入 RTC 的驱动支持，依次选择如下所示的菜单项。

```
Device Drivers --->
<*> Real Time Clock --->
```

在此可以看到在默认配置中已经选择了 RTC 相关的选项，读者需要注意的是，该配置菜单最下方的<*>



Samsung S3C series SoC RTC 选项支持，因为这里才是内核中真正的 2440 的 RTC 驱动配置项。

（3）测试 RTC

退出内核配置菜单，执行如下所示的命令。

```
#make zImage
```

把生成的 arch/arm/boot/zImage 烧写到开发板后，即可在/dev 目录下看到/dev/rtc 设备驱动。要想测试 RTC，可以参考 mini2440 用户手册的内容。在 Linux 中一般使用 date 命令来更改时间的方法，为了把 S3C2440 内部带的时钟与 Linux 系统时钟同步，一般使用 hwclock 命令来实现，下面是它们的使用方法。

- ☑ date -s 042916352014 #：功能是设置时间为 2014-04-29 16:34。
- ☑ hwclock -w #：把刚刚设置的时间存入 S3C2440 内部的 RTC。
- ☑ 开机时使用“hwclock –s”命令可以恢复 Linux 的系统时钟为 RTC，一般把该语句放入/etc/init.d/rcS 文件中自动执行。

启动设备后即可看到 dev 目录下的 RTC 设备的设备文件，如图 15-3 所示。

```
ptyv0                 ttyp9                zero
ptyv1                 ttypa
[root@FriendlyARM /dev]#
[root@FriendlyARM /dev]#
[root@FriendlyARM /dev]# ls -l|grep rtc
crw-r--r--    1 root     root     254,   0 Jan  1 08:00 rtc
lrwxrwxrwx    1 root     root            3 Jan  1 08:00 rtc0 -> rtc
[root@FriendlyARM /dev]#
```

图 15-3　输出信息

15.7.4　实现一个秒表定时器

本节将通过一个渠道程序启动一个系统定时器，这个定时器以 1 秒为间隔不断地调用定时器处理函数。每调用函数一次，计数器就会加 1。调用设备文件 dev/timer_demo 中的函数 read()，可以读取定时器的值。

（1）驱动程序文件 timer_demo.c 的具体实现代码如下。

```
#include <linux/module.h>
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/mm.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <asm/io.h>
#include <asm/system.h>
#include <asm/uaccess.h>
#include <linux/timer.h>
#include <asm/atomic.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/delay.h>
#define DEVICE_NAME "timer_demo"
struct timer_dev
{
	atomic_t counter;	//一共经历多少秒
	struct timer_list s_timer;	//设备要使用的定时器
};
struct timer_dev *timer_devp;	//设备结构体指针
//定时器处理函数
static void timer_demo_handle(unsigned long arg)
{
	//使定时器处理函数可以在下一秒执行
	mod_timer(&timer_devp->s_timer, jiffies + HZ);
```

```
        atomic_inc(&timer_devp->counter);
        printk(KERN_NOTICE "current jiffies is %ld\n", jiffies);
}
//设备文件打开函数
int timer_demo_open(struct inode *inode, struct file *filp)
{
        //初始化定时器
        init_timer(&timer_devp->s_timer);
        timer_devp->s_timer.function = &timer_demo_handle;
        timer_devp->s_timer.expires = jiffies + HZ;
        add_timer(&timer_devp->s_timer); //添加（注册）定时器
        atomic_set(&timer_devp->counter, 0);
        //计数清 0
        return 0;
}
//设备文件释放函数
int timer_demo_release(struct inode *inode, struct file *filp)
{
        del_timer_sync(&timer_devp->s_timer);
        return 0;
}
//设备文件的读函数
static ssize_t timer_demo_read(struct file *filp, char __user *buf,
              size_t count, loff_t *ppos)
{
        int counter;
        counter = atomic_read(&timer_devp->counter);
        if (put_user(counter, (int*)buf))
                return -EFAULT;
        else
                return sizeof(unsigned int);
}

/*文件操作结构体*/
static const struct file_operations timer_fops =
{ .owner = THIS_MODULE, .open = timer_demo_open, .release = timer_demo_release,
              .read = timer_demo_read };
static struct miscdevice misc =
{ .minor = MISC_DYNAMIC_MINOR, .name = DEVICE_NAME, .fops = &timer_fops };
/*设备驱动模块加载函数*/
int __init timer_demo_init(void)
{
        int ret = misc_register(&misc);
        /*动态申请设备结构体的内存*/
        timer_devp = kmalloc(sizeof(struct timer_dev), GFP_KERNEL);
        if (!timer_devp) /*申请失败*/
        {
                misc_deregister(&misc);
                ret -ENOMEM;
        }
        else
```

```
	{
		memset(timer_devp, 0, sizeof(struct timer_dev));
	}
	return ret;
}

/*模块卸载函数*/
void __exit timer_demo_exit(void)
{
	kfree(timer_devp); //释放设备结构体内存
	misc_deregister(&misc);
}
MODULE_AUTHOR("Lining");
MODULE_LICENSE("GPL");
module_init(timer_demo_init);
module_exit(timer_demo_exit);
```

（2）编写驱动测试程序 test_timer.c，具体实现代码如下。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/time.h>

main()
{
  int fd;
  int counter = 0;
  int old_counter = 0;

  /*打开/dev/timer_demo 设备文件*/
  fd = open("/dev/timer_demo", O_RDONLY);
  if (fd !=   - 1)
  {
    while (1)
    {
      read(fd,&counter, sizeof(unsigned int));
      if(counter!=old_counter)
      {
        printf("seconds after open /dev/timer_demo :%d\n",counter);
        old_counter = counter;
      }
    }
  }
  else
  {
    printf("Device open failure\n");
  }
}
```

执行后将会在 Linux 终端输出驱动计数器的值。

第 16 章　振动器驱动架构和移植

振动器是 Android 智能手机操作系统中比较常见的功能之一，在实际应用中可以将来电选项设置为振动模式作为提醒。在 Android 系统中，通过振动系统模块可以实现来电铃声和闹钟的振动功能。本章将详细讲解 Android 振动器系统驱动的实现和移植内容，为读者学习本书后面的知识打下基础。

16.1　振动器系统架构

在 Android 系统中，振动器是负责控制启动或关闭电话振动功能的设备。Android 系统中的振动系统包括驱动程序、硬件抽象层、JNI 部分、Java 框架类等部分，并且向 Java 应用程序层提供了简单的 API 作为平台接口。Android 振动器系统的基本层次结构如图 16-1 所示。

Android 振动器系统自下而上包含了驱动程序、振动器系统硬件抽象层、振动器系统 Java 框架类、Java 框架中振动器系统使用等几个部分，其结构如图 16-2 所示。

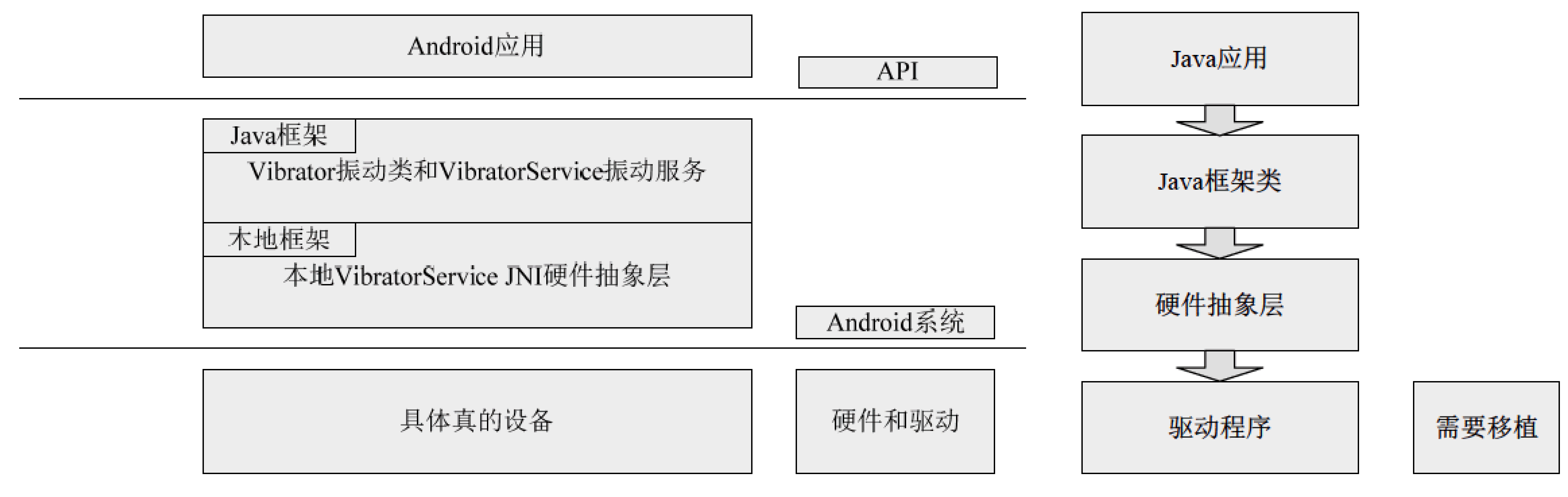


图 16-1　Android 振动器系统的框架结构　　　图 16-2　振动器系统结构元素

在图 16-2 中，各个构成元素的具体说明如下。

（1）驱动程序

驱动程序是某特定硬件平台振动器的驱动程序，通常基于 Android 的 Timed Output 驱动框架来实现。

（2）硬件抽象层

振动器系统的硬件抽象层接口路径是 hardware/libhardware_legacy/include/hardware_legacy/vibrator.h。

振动系统的硬件抽象层的默认代码路径是 hardware/libhardware_legacy/vibrator/vibrator.c。

因为 Android 振动器的硬件抽象层是 libhardware_legacy.so 的一部分，所以通常并不需要重新实现。

（3）JNI 框架部分

振动器系统的 JNI 框架部分的代码路径是 frameworks/base/services/jni/com_android_server_VibratorService.cpp。

在此文件中定义了振动器的 JNI 部分，通过调用硬件抽象层向上层提供接口。

（4）Java 应用部分

振动器系统的 Java 部分的代码路径是 frameworks/base/services/java/com/android/server/VibratorService.java

和 frameworks/base/core/java/android/os/Vibrator.java。

文件 VibratorService.java 通过调用 VibratorService JNI 来实现包 com.android.server 中的类 VibratorService。类 VibratorService 不是平台的 API，只被 Android 系统 Java 框架中的一小部分调用。

在文件 Vibrator.java 中实现了 android.os 包中的 Vibrator 类，这是向 Java 层提供的 API。

16.2　硬件抽象层架构

在 Android 系统中，振动器系统的硬件抽象层接口的实现文件是 vibrator.h，被保存在目录 hardware/libhardware_legacy/include/hardware_legacy/中。

文件 vibrator.h 的主要实现代码如下所示。

```
#ifndef _HARDWARE_VIBRATOR_H
#define _HARDWARE_VIBRATOR_H
 #if __cplusplus
 extern "C" {
 #endif

 /**
    * 开始振动
   *
   * @振动时间，单位毫秒
   *
   * @返回 0 表示成功，返回 1 表示出错
   */
 int vibrator_on(int timeout_ms);

 /**
   * 关闭 vibrator
 *
   * @返回 0 表示成功，返回 1 表示出错
   */
 int vibrator_off();

 #if __cplusplus
 }  // extern "C"
#endif
#endif  // _HARDWARE_VIBRATOR_H
```

在 Android 系统中，振动系统的硬件抽象层的默认代码路径是 hardware/libhardware_legacy/vibrator/vibrator.c。

文件 vibrator.c 的主要代码如下所示。

```
int vibrator_exists()
{
    int fd;

#ifdef QEMU_HARDWARE
    if (qemu_check()) {
        return 1;
    }
```

```
#endif

        fd = open(THE_DEVICE, O_RDWR);
        if(fd < 0)
                return 0;
        close(fd);
        return 1;
}

static int sendit(int timeout_ms)
{
        int nwr, ret, fd;
        char value[20];

#ifdef QEMU_HARDWARE
        if (qemu_check()) {
                return qemu_control_command( "vibrator:%d", timeout_ms );
        }
#endif

        fd = open(THE_DEVICE, O_RDWR);
        if(fd < 0)
                return errno;

        nwr = sprintf(value, "%d\n", timeout_ms);
        ret = write(fd, value, nwr);

        close(fd);

        return (ret == nwr) ? 0 : -1;
}

int vibrator_on(int timeout_ms)
{
        /* constant on, up to maximum allowed time */
        return sendit(timeout_ms);
}

int vibrator_off()
{
        return sendit(0);
}
```

16.3　JNI 层架构

在 Android 系统中，振动器系统的 JNI 框架部分的实现文件是 com_android_server_VibratorService.cpp，主要实现代码如下所示。



```
namespace android
{

static jboolean vibratorExists(JNIEnv *env, jobject clazz)
{
    return vibrator_exists() > 0 ? JNI_TRUE : JNI_FALSE;
}

static void vibratorOn(JNIEnv *env, jobject clazz, jlong timeout_ms)
{
    // ALOGI("vibratorOn\n");
    vibrator_on(timeout_ms);
}

static void vibratorOff(JNIEnv *env, jobject clazz)
{
    // ALOGI("vibratorOff\n");
    vibrator_off();
}

static JNINativeMethod method_table[] = {
    { "vibratorExists", "()Z", (void*)vibratorExists },
    { "vibratorOn", "(J)V", (void*)vibratorOn },//振动器开
    { "vibratorOff", "()V", (void*)vibratorOff }//振动器关
};

int register_android_server_VibratorService(JNIEnv *env)
{
    return jniRegisterNativeMethods(env, "com/android/server/VibratorService",
            method_table, NELEM(method_table));
}

};
```

在上述代码中，核心功能是通过 JNINativeMethod method_table[]和 register_android_server_VibratorService()实现的。

16.4　Java 层架构

在 JNI 层文件 com_android_server_VibratorService.cpp 中，调用了 VibratorService JNI 来实现包 com.android.server 中的 VibratorService 类。类 VibratorService 在文件 frameworks/base/services/java/com/android/server/VibratorService.java 中定义，主要实现代码如下所示。

```
//系统准备
    public void systemReady() {
        mIm = (InputManager)mContext.getSystemService(Context.INPUT_SERVICE);

        mContext.getContentResolver().registerContentObserver(
                Settings.System.getUriFor(Settings.System.VIBRATE_INPUT_DEVICES), true,
                new ContentObserver(mH) {
```

```
                    @Override
                    public void onChange(boolean selfChange) {
                        updateInputDeviceVibrators();
                    }
                }, UserHandle.USER_ALL);

        mContext.registerReceiver(new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {
                updateInputDeviceVibrators();
            }
        }, new IntentFilter(Intent.ACTION_USER_SWITCHED), null, mH);

        updateInputDeviceVibrators();
    }
//已经开始振动
    public boolean hasVibrator() {
        return doVibratorExists();
    }
//验证传入的 UID
    private void verifyIncomingUid(int uid) {
        if (uid == Binder.getCallingUid()) {
            return;
        }
        if (Binder.getCallingPid() == Process.myPid()) {
            return;
        }
        mContext.enforcePermission(android.Manifest.permission.UPDATE_APP_OPS_STATS,
                Binder.getCallingPid(), Binder.getCallingUid(), null);
    }
//振动函数，必须确保在服务器发生振动，不能超时
    public void vibrate(int uid, String packageName, long milliseconds, IBinder token) {
        if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.VIBRATE)
                != PackageManager.PERMISSION_GRANTED) {
            throw new SecurityException("Requires VIBRATE permission");
        }
        verifyIncomingUid(uid);
        // We're running in the system server so we cannot crash. Check for a
        // timeout of 0 or negative. This will ensure that a vibration has
        // either a timeout of > 0 or a non-null pattern.
        if (milliseconds <= 0 || (mCurrentVibration != null
                && mCurrentVibration.hasLongerTimeout(milliseconds))) {
            // Ignore this vibration since the current vibration will play for
            // longer than milliseconds
            return;
        }

        Vibration vib = new Vibration(token, milliseconds, uid, packageName);

        final long ident = Binder.clearCallingIdentity();
        try {
```

```
                synchronized (mVibrations) {
                    removeVibrationLocked(token);
                    doCancelVibrateLocked();
                    mCurrentVibration = vib;
                    startVibrationLocked(vib);
                }
            } finally {
                Binder.restoreCallingIdentity(ident);
            }
        }
//设置振动模式函数，可以设置振动重复，也可以设置振动时长
        public void vibratePattern(int uid, String packageName, long[] pattern, int repeat,
                IBinder token) {
            if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.VIBRATE)
                    != PackageManager.PERMISSION_GRANTED) {
                throw new SecurityException("Requires VIBRATE permission");
            }
            verifyIncomingUid(uid);
            // so wakelock calls will succeed
            long identity = Binder.clearCallingIdentity();
            try {
                if (false) {
                    String s = "";
                    int N = pattern.length;
                    for (int i=0; i<N; i++) {
                        s += " " + pattern[i];
                    }
                    Slog.i(TAG, "vibrating with pattern: " + s);
                }

                // we're running in the server so we can't fail
                if (pattern == null || pattern.length == 0
                        || isAll0(pattern)
                        || repeat >= pattern.length || token == null) {
                    return;
                }

                Vibration vib = new Vibration(token, pattern, repeat, uid, packageName);
                try {
                    token.linkToDeath(vib, 0);
                } catch (RemoteException e) {
                    return;
                }

                synchronized (mVibrations) {
                    removeVibrationLocked(token);
                    doCancelVibrateLocked();
                    if (repeat >= 0) {
                        mVibrations.addFirst(vib);
                        startNextVibrationLocked();
                    } else {
```

```
                    // A negative repeat means that this pattern is not meant
                    // to repeat. Treat it like a simple vibration
                    mCurrentVibration = vib;
                    startVibrationLocked(vib);
                }
            }
        }
        finally {
            Binder.restoreCallingIdentity(identity);
        }
    }
//取消振动
    public void cancelVibrate(IBinder token) {
        mContext.enforceCallingOrSelfPermission(
                android.Manifest.permission.VIBRATE,
                "cancelVibrate");

        // so wakelock calls will succeed
        long identity = Binder.clearCallingIdentity();
        try {
            synchronized (mVibrations) {
                final Vibration vib = removeVibrationLocked(token);
                if (vib == mCurrentVibration) {
                    doCancelVibrateLocked();
                    startNextVibrationLocked();
                }
            }
        }
        finally {
            Binder.restoreCallingIdentity(identity);
        }
    }
```

然后通过文件 frameworks/base/core/java/android/os/Vibrator.java 实现包 android.os 中的 Vibrator 类，然后获得名称为 vibrator 的服务，并配合目录中的 IVibratorService.aidl 文件向应用程序层提供 Vibrator 的相关 API。文件 Vibrator.java 的主要实现代码如下所示。

```
package android.os;
import android.content.Context;
public abstract class Vibrator {
    public Vibrator() {
    }
    /**
     *检查硬件是否有一个振动器
     */
    public abstract boolean hasVibrator();

    /**
     *在指定的时间一直振动
     */
    public abstract void vibrate(long milliseconds);

    /**
```

```
        *对于一个给定的模式振动
        */
       public abstract void vibrate(long[] pattern, int repeat);
       public abstract void vibrate(int owningUid, String owningPackage, long milliseconds);

       public abstract void vibrate(int owningUid, String owningPackage, long[] pattern, int repeat);

       /**
        *  关闭 vibrator
        */
       public abstract void cancel();
}
```

16.5　实战演练——移植振动器系统

在 Android 底层开发应用中，根据特定的硬件平台有如下两种移植振动器系统的方法。

（1）由于已经具有硬件抽象层，振动器系统的移植只需要实现驱动程序即可，这个驱动程序需要基于 Android 内核中的 Timed Output 驱动框架。

（2）根据自己实现的驱动程序，在 libhardware_legacy.so 库中重新实现振动器的硬件抽象层定义接口。因为振动器硬件抽象层的接口非常简单，所以这种实现方式非常简单。

本节将详细讲解振动器系统的具体移植方法。

16.5.1　移植振动器驱动程序

要想实现 Vibrator 驱动程序，只需要实现振动的接口即可。因为这是一个输出设备，所以需要接收振动时间作为参数。在 Android 中，可以使用多种方式来实现 Vibrator 驱动程序。在此推荐基于 Android 内核定义的 Timed Output 驱动程序框架实现 Vibrator 的驱动程序。

Timed Output 有“定时输出”之意，用于定时发出某个输出，此种驱动程序依然是基于 sys 文件系统来完成的。

在文件 drivers/staging/android/timed_output.h 中定义了一个名为 timed_output_dev 的结构体，其中包含了 enable 和 get_time 两个函数指针，当实现结构体后，使用函数 timed_output_dev_register()实现注册，使用函数 timed_output_dev_unregister()实现注销操作。

Timed Output 驱动程序框架将为每个设备在/sys/class/timed_output/目录中建立一个子目录，其中设备子目录中的 enable 文件就是设备的控制文件。当读这个 enable 文件时表示获得剩余时间，当写这个文件时表示根据时间振动。虽然 Timed Output 类型驱动本身有获得剩余时间的能力（读 enable 文件），但是在 Android Vibrator 硬件抽象层以上的各层接口都没有使用这个功能。

通过 sys 文件系统可以调试 Timed Output 驱动设备，对于 Vibrator 设备来说，其实现的 Timed Output 驱动程序的名称是 vibrator，所以 Vibrator 设备在 sys 文件系统中的方法如下所示。

```
# echo "10000" > /sys/class/timed_output/vibrator/enable
# cat /sys/class/timed_output/vibrator/enable
3290
# echo "0"   > /sys/class/timed_output/vibrator/enable
```

对于 enable 文件来说，“写”表示使能指定的时间，“读”表示获取剩余时间。

16.5.2　实现硬件抽象层

下面将详细讲解实现硬件抽象层的具体流程。

1．硬件抽象层的接口

由前面的知识可以了解到，Vibrator 硬件抽象层的接口在文件 hardware/libhardware_legacy/include/hardware_legacy/vibrator.h 中。

文件 vibrator.h 的核心代码如下所示。

```
int vibrator_on(int timeout_ms);                    //开始振动
int vibrator_off();                                 //关闭振动
```

在文件 vibrator.h 中定义了两个接口，分别表示振动和关闭，振动开始以毫秒（ms）作为时间单位。

2．实现标准硬件抽象层

Vibrator 硬件抽象层是标准的实现代码，定义在文件 hardware/libhardware_legacy/vibrator/vibrator.c 中。

文件 vibrator.c 的核心内容是函数 sendit()，此函数的实现代码如下所示。

```
#define THE_DEVICE "/sys/class/timed_output/vibrator/enable"
static int sendit(int timeout_ms)
{
    int nwr, ret, fd;
    char value[20];
#ifdef QEMU_HARDWARE                          //使用 QEMU 的情况
    if (qemu_check()) {
        return qemu_control_command( "vibrator:%d", timeout_ms );
    }
#endif
    fd = open(THE_DEVICE, O_RDWR);
//读取 sys 文件系统中的内容
    if(fd < 0) return errno;
    nwr = sprintf(value, "%d\n", timeout_ms);
    ret = write(fd, value, nwr);
    close(fd);
    return (ret == nwr) ? 0 : -1;
}
```

上述 sendit()函数的功能是根据时间进行“振动”，在真实的硬件中是通过 sys 文件系统的文件进行控制的，如果是模拟器环境则通过 QEMU 发送命令。其中 vibrator_on()调用 sendit()以时间作为参数，vibrator_on()调用 sendit()以 0 作为参数。

16.6　实战演练——在 MSM 平台实现振动器驱动

在 MSM 的 Mahimahi 平台中，因为是基于 Timed Output 驱动程序框架的驱动程序来实现 Vibrator 的，所以不需要再实现硬件抽象层。在 Mahimahi 平台中，Vibrator 驱动程序在内核文件 arch/arm/mach-msm/msm_vibrator.c 中实现。

文件 msm_vibrator.c 的实现代码如下所示。



```
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/err.h>
#include <linux/hrtimer.h>
#include <../../../drivers/staging/android/timed_output.h>
#include <linux/sched.h>

#include <mach/msm_rpcrouter.h>

#define PM_LIBPROG          0x30000061
#if (CONFIG_MSM_AMSS_VERSION == 6220) || (CONFIG_MSM_AMSS_VERSION == 6225)
#define PM_LIBVERS          0xfb837d0b
#else
#define PM_LIBVERS          0x10001
#endif

#define HTC_PROCEDURE_SET_VIB_ON_OFF          21
#define PMIC_VIBRATOR_LEVEL     (3000)

static struct work_struct work_vibrator_on;
static struct work_struct work_vibrator_off;
static struct hrtimer vibe_timer;

static void set_pmic_vibrator(int on)
{
        static struct msm_rpc_endpoint *vib_endpoint;
        struct set_vib_on_off_req { /*  定义 RPC 的端点  */
                struct rpc_request_hdr hdr;
                uint32_t data;
        } req;
        if (!vib_endpoint) {
                vib_endpoint = msm_rpc_connect(PM_LIBPROG, PM_LIBVERS, 0);
                if (IS_ERR(vib_endpoint)) {
                        printk(KERN_ERR "init vib rpc failed!\n");
                        vib_endpoint = 0;
                        return;
                }
        }
        if (on)
                req.data = cpu_to_be32(PMIC_VIBRATOR_LEVEL);
/*  得到请求时间  */
        else
                req.data = cpu_to_be32(0);
        msm_rpc_call(vib_endpoint, HTC_PROCEDURE_SET_VIB_ON_OFF, &req,
                sizeof(req), 5 * HZ); /*  进行 RPC 调用  */
}

static void pmic_vibrator_on(struct work_struct *work)
{
        set_pmic_vibrator(1);
```

```
}

static void pmic_vibrator_off(struct work_struct *work)
{
        set_pmic_vibrator(0);
}

static void timed_vibrator_on(struct timed_output_dev *sdev)
{
        schedule_work(&work_vibrator_on);
}

static void timed_vibrator_off(struct timed_output_dev *sdev)
{
        schedule_work(&work_vibrator_off);
}

static void vibrator_enable(struct timed_output_dev *dev, int value)
{
        hrtimer_cancel(&vibe_timer);

        if (value == 0)
                timed_vibrator_off(dev);
        else {
                value = (value > 15000 ? 15000 : value);
                timed_vibrator_on(dev);
                hrtimer_start(&vibe_timer,
                                ktime_set(value / 1000, (value % 1000) * 1000000),
                                HRTIMER_MODE_REL);
        }
}

static int vibrator_get_time(struct timed_output_dev *dev)
{
        if (hrtimer_active(&vibe_timer)) {
                ktime_t r = hrtimer_get_remaining(&vibe_timer);
                return r.tv.sec * 1000 + r.tv.nsec / 1000000;
        } else
                return 0;
}

static enum hrtimer_restart vibrator_timer_func(struct hrtimer *timer)
{
        timed_vibrator_off(NULL);
        return HRTIMER_NORESTART;
}

static struct timed_output_dev pmic_vibrator = {
        .name = "vibrator",
        .get_time = vibrator_get_time,
```

```
        .enable = vibrator_enable,
};

void __init msm_init_pmic_vibrator(void)
{
        INIT_WORK(&work_vibrator_on, pmic_vibrator_on);
        INIT_WORK(&work_vibrator_off, pmic_vibrator_off);
        hrtimer_init(&vibe_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL); /* 定时器 */
        vibe_timer.function = vibrator_timer_func;
        timed_output_dev_register(&pmic_vibrator);              /* 注册 timed_output_dev 设备 */
}
MODULE_DESCRIPTION("timed output pmic vibrator device");
MODULE_LICENSE("GPL");
```

在上述实现代码中，核心功能是通过函数 set_pmic_vibrator()实现的，此函数通过 MSM 系统的远程过程调用（RPC）实现了具体的功能，调用的指令由 HTC_PROCEDURE_SET_VIB_ON_OFF 指定。

上述 MSM 驱动程序的初始化处理是通过上述代码中的函数 msm_init_pmic_vibrator(void)实现的，其中 vibrator_work 为 work_struct 类型，在队列的执行函数 update_vibrator()中调用函数 set_pmic_vibrator()。

pmic_vibrator 是一个 timed_output_dev 类型的设备，其 enable 指针的实现 vibrator_enable 会根据输入的数值开始定时，并通过向调度队列进行输出操作。函数 get_time()指针的实现 vibrator_get_time 只能从定时器中获取剩余时间。此处之所以使用定时器加队列的方式，是因为 enable 的调用将形成一个持续时间的效果，但是调用本身并不宜阻塞，所以在实现中让函数 vibrator_enable()退出后通过定时器实现效果。

16.7　实战演练——在 MTK 平台实现振动器驱动

下面将详细讲解在 MTK6573 平台实现振动器驱动的方法。

（1）实现驱动层

修改如下 Android 系统中新增的 Linux 内核文件。

☑　/kernel/drivers/staging/android/timed_output.h

☑　/kernel/drivers/staging/android/timed_output.c

在文件 timed_output.h 中定义结构体 timed_output_dev，具体实现代码如下所示。

```
struct timed_output_dev {
    const char    *name;
    /* enable the output and set the timer */
    void      (*enable)(struct timed_output_dev *sdev, int timeout);

    /* returns the current number of milliseconds remaining on the timer */
    int       (*get_time)(struct timed_output_dev *sdev);

    /* private data */
    struct device     *dev;
    int       index;
    int       state;
};
```

在文件 timed_output.c 中实现 timed_output_dev 结构体，使用函数 timed_output_dev_register()实现注册，使用 timed_output_dev_unregister 实现注销。

```
int timed_output_dev_register(struct timed_output_dev *tdev)
{
	int ret;

	if (!tdev || !tdev->name || !tdev->enable || !tdev->get_time)
		return -EINVAL;

	ret = create_timed_output_class();
	if (ret < 0)
		return ret;

	tdev->index = atomic_inc_return(&device_count);
	tdev->dev = device_create(timed_output_class, NULL,
		MKDEV(0, tdev->index), NULL, tdev->name);
	if (IS_ERR(tdev->dev))
		return PTR_ERR(tdev->dev);

	ret = device_create_file(tdev->dev, &dev_attr_enable);
	if (ret < 0)
		goto err_create_file;

	dev_set_drvdata(tdev->dev, tdev);
	tdev->state = 0;
	return 0;

err_create_file:
	device_destroy(timed_output_class, MKDEV(0, tdev->index));
	printk(KERN_ERR "timed_output: Failed to register driver %s\n",
			tdev->name);

	return ret;
}
EXPORT_SYMBOL_GPL(timed_output_dev_register);

void timed_output_dev_unregister(struct timed_output_dev *tdev)
{
	device_remove_file(tdev->dev, &dev_attr_enable);
	device_destroy(timed_output_class, MKDEV(0, tdev->index));
	dev_set_drvdata(tdev->dev, NULL);
}
EXPORT_SYMBOL_GPL(timed_output_dev_unregister);
```

（2）驱动实现移植

在 MTK6573 平台中实现文件是./mediatek/platform/mt6573/kernel/drivers/vibrator/vibrator.c。首先打开手机调试并连接 USB，执行 adb shell 命令进入目录/sys/devices/timed_output/vibrator/。

执行“echo "10000" enable”后会发现手机在振动。

```
# echo "10000" enable
echo "10000" enable
10000 enable
```

执行“cat enable”命令可以查看当前振动时间的剩余数。



```
# cat enable
cat enable
0
```

（3）实现硬件抽象层

Android 系统将对底层驱动的调用封装成为硬件抽象层，实现文件是/hardware/libhardware_legacy/vibrator/vibrator.c。

具体实现代码如下所示。

```
int vibrator_on(int timeout_ms)
{
    /* constant on, up to maximum allowed time */
    return sendit(timeout_ms);
}

int vibrator_off()
{
    return sendit(0);
}
```

（4）实现 JNI 框架层

Android JNI 框架层是为方便 Java 调用 C/C++定义的方法，具体实现文件是./frameworks/base/services/jni/com_android_server_VibratorService.cpp。

具体实现代码如下所示。

```
namespace android
 {

 static void vibratorOn(JNIEnv *env, jobject clazz, jlong timeout_ms)
 {
     // LOGI("vibratorOn\n");
     vibrator_on(timeout_ms);
 }
 static void vibratorOff(JNIEnv *env, jobject clazz)
 {
     // LOGI("vibratorOff\n");
     vibrator_off();
 }
 static JNINativeMethod method_table[] = {
     { "vibratorOn", "(J)V", (void*)vibratorOn },
     { "vibratorOff", "()V", (void*)vibratorOff }
 };
 int register_android_server_VibratorService(JNIEnv *env)
 {
     return jniRegisterNativeMethods(env, "com/android/server/VibratorService",
             method_table, NELEM(method_table));
 }
};
```

（5）实现 Java 应用层

Java 应用层包括 Java 应用的调用和 Android 系统服务 Java 层，具体实现文件是./frameworks/base/services/java/com/android/server/VibratorService.java。

读者可以编写 Java 应用文件来测试我们的驱动程序。

16.8　实战演练——移植振动器驱动

因为在 Android 系统中已经实现了底层驱动 timed_gpio，把定时功能和 gpio 的功能结合在一起。振动器犹如一个小直流电机，当 gpio 口是高电平时会转动电机，电机在 gpio 口为低电平时就不转。其中 time 是控制转的时间，也就是 gpio 口处于高电平的时间。移植振动器驱动的具体流程如下所示。

（1）底层驱动 timed_gpio 的具体代码在文件/drivers/staging/android/timed_gpio.c 中实现，在移植振动器时只需在相关平台的 platform.c 文件中加入 platform device 即可完成，具体实现代码如下所示。

```
static struct timed_gpio vibrator =
{
        .name = “vibrator”,
        .gpio = 61, //对应自己平台的 gpio 号
        .max_timeout = 100000,
        .active_low = 0;
};
static   struct timed_gpio_platform_data timed_gpio_data =
{
        .num_gpios = 1,
        .gpios = &vibrator,
};
static struct platform_device my_timed_gpio =

{
        .name = “timed-gpio”,
        .id = -1,
        .dev =
        {
                .platform_data = &timed_gpio_data,
        },
};
```

（2）在 make menuconfig 命令中，选择设备中 staging 下的 Android 中的相关选项，如图 16-3 所示。

```
[*] Android Drivers
[*]   Android Binder IPC Driver
<*>   Android log driver
[*]   Android RAM buffer console
[*]     Enable verbose console messages on Android RAM console
[*]     Start Android RAM console early
(0)       Android RAM console virtual address
(0)       Android RAM console buffer size
[*]   Timed output class driver
<*>     Android timed gpio driver
[*]   Android Low Memory Killer
```

图 16-3　选择相关选项

（3）接下来编译运行内核，在内核运行后可以进行测试工作。此时 timed gpio 驱动程序会为每个设备在/sys/class/timed_output/目录下建立一个子目录，设备子目录中的 enable 文件就是控制设备的时间。因为在 platform 中名称为 vibrator，所以可以用以下命令进行测试。

```
echo 10000 > /sys/class/timed_output/vibrator/enable
```

（4）接下来可以看到振动器在转动，此时也可以用示波器或者万用表来验证。接着执行下面的命令。

```
cat /sys/class/timed_output/vibrator/enable
```

此时会发现 enable 的值一直在变小，直到为 0 时停止转动。

到此为止，整个底层驱动测试完毕。而 Android 上层就会容易很多，因为上层几乎和平台关系不大，所以要修改的东西不多。



第 17 章　输入系统驱动

Android 输入系统的结构比较简单，实现输入功能的硬件设备包括键盘、触摸屏和轨迹球等。在 Android 的上层中，可以获得这些设备产生的事件并对设备的事件做出响应。在 Java 框架中，通常使用运动事件来获得触摸屏和轨迹球设备的信息，使用按键事件获得各种键盘的信息。本章将详细讲解 Android 输入系统驱动架构的基本知识，为读者学习本书后面的知识打下基础。

17.1　输入系统架构分析

Android 输入系统的基本框架结构如图 17-1 所示。

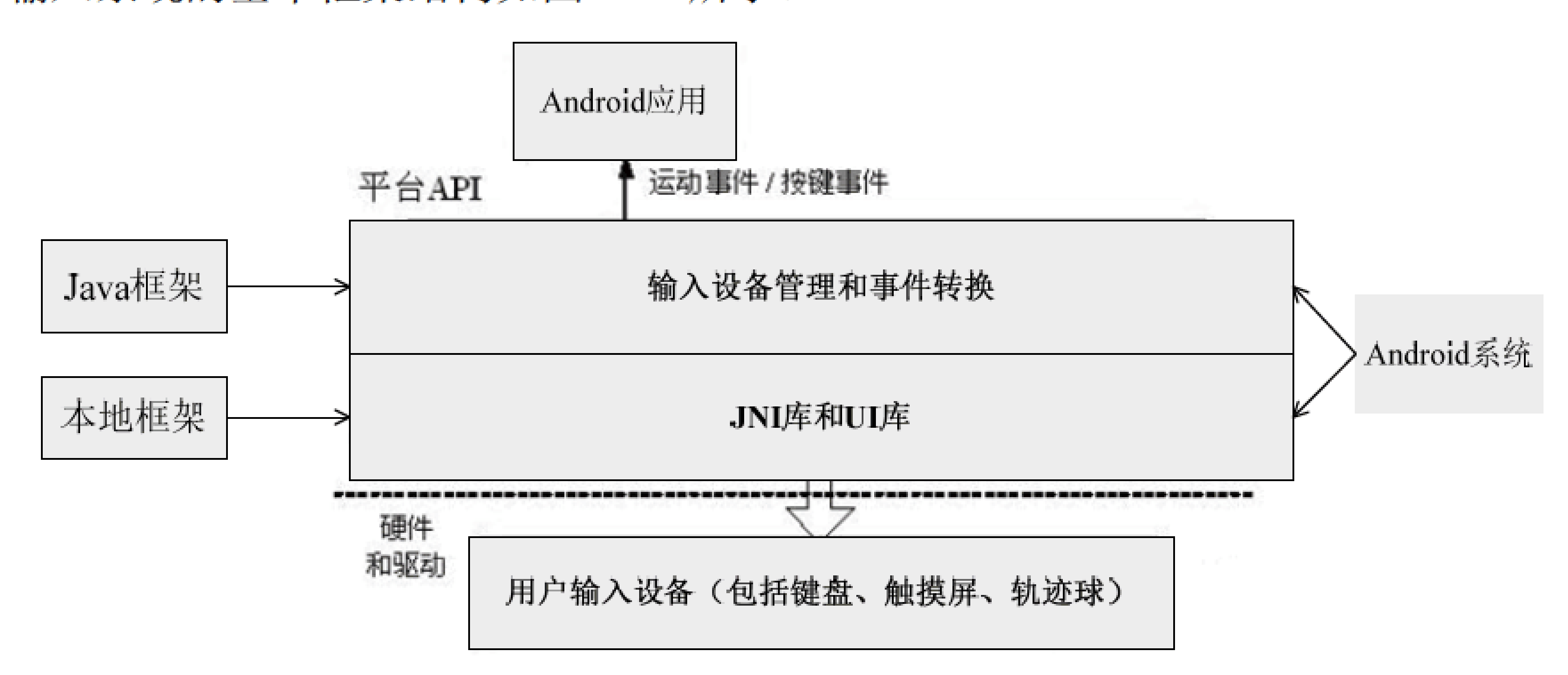


图 17-1　Android 输入系统的框架结构

Android 输入系统的结构比较简单，自下而上包含了驱动程序、本地库处理部分、Java 类对输入事件的处理和 Java 程序的接口等，如图 17-2 所示。

在图 17-2 中，从上到下各个结构元素的具体说明如下所示。

（1）Android 应用程序层

在 Android 系统的应用程序层中，通过重新实现 onTouchEvent()和 onTrackballEvent()等函数来接收运动事件（MotionEvent），通过重新实现 onKeyDown()和 onKeyUp()等函数来接收按键事件（KeyEvent）。这些类包含在 android.view 包中。

（2）Java 框架层的处理

在 Android 系统的 Java 框架层中，通过 KeyInputDevice 等类处理由 EventHub 传送上来的信息，这些信息通常由数据结构 RawInputEvent 和 KeyEvent 来表示。在一般情况下，对于按键事件直接使用 KeyEvent 来传送给应用程序层。对于触摸屏和轨迹球等事件，则由 RawInputEvent 经过转换后形成 MotionEvent 时间传送给应用程序层。

（3）EventHub

在 Android 系统中，本地框架层的 EventHub 是 libui 中的一部分，它实现了对驱动程序的控制，并从中获得信息。定义按键布局和按键字符映射需要运行时配置文件的支持，它们的后缀名分别为 kl 和 kcm。

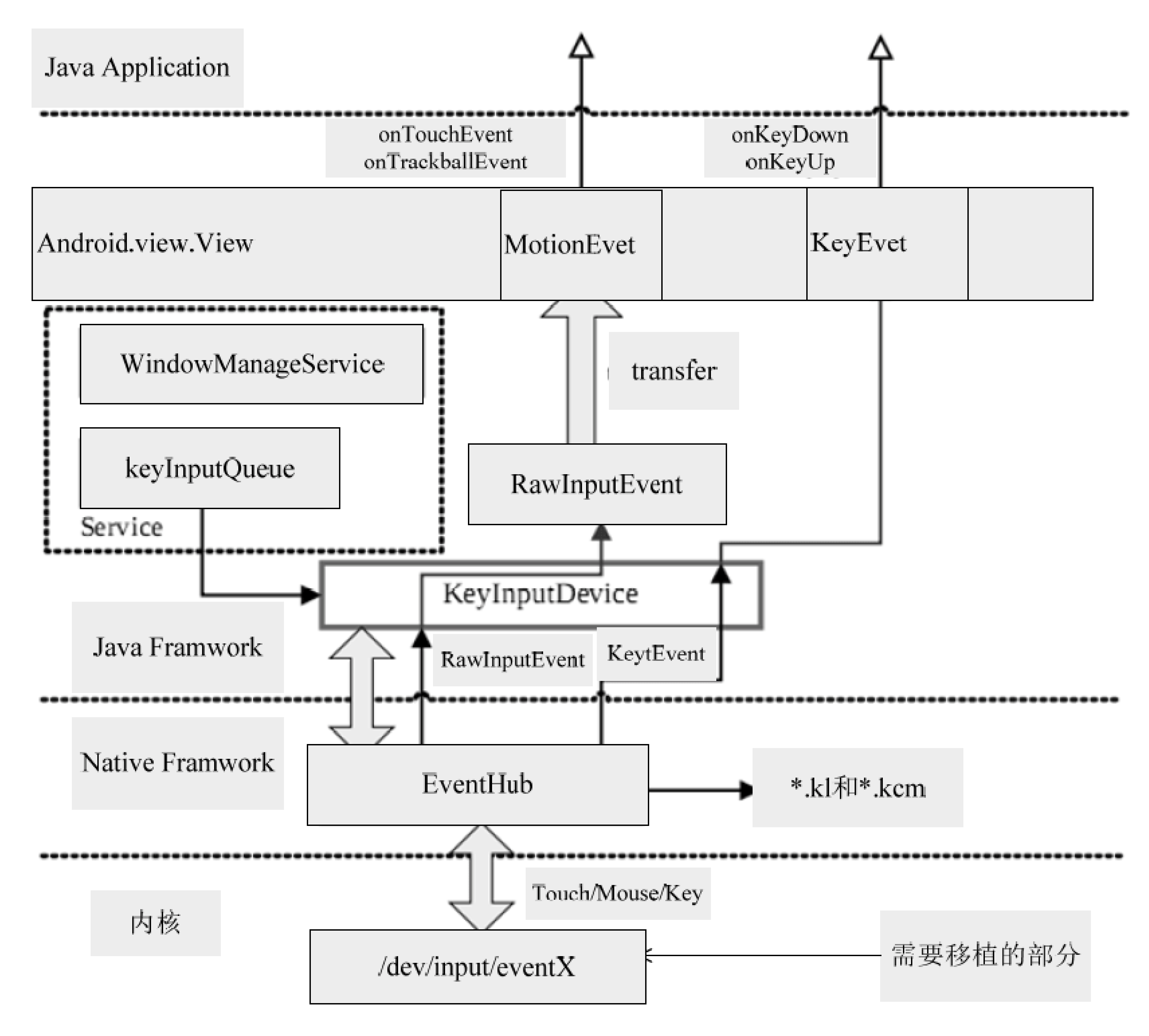


图 17-2　用户输入系统的结构

（4）驱动程序

在 Android 系统中，输入系统的驱动程序保存在/dev/input 目录中，通常是 Event 类型的驱动程序。在 Android 系统中，Input 输入子系统的架构如图 17-3 所示。

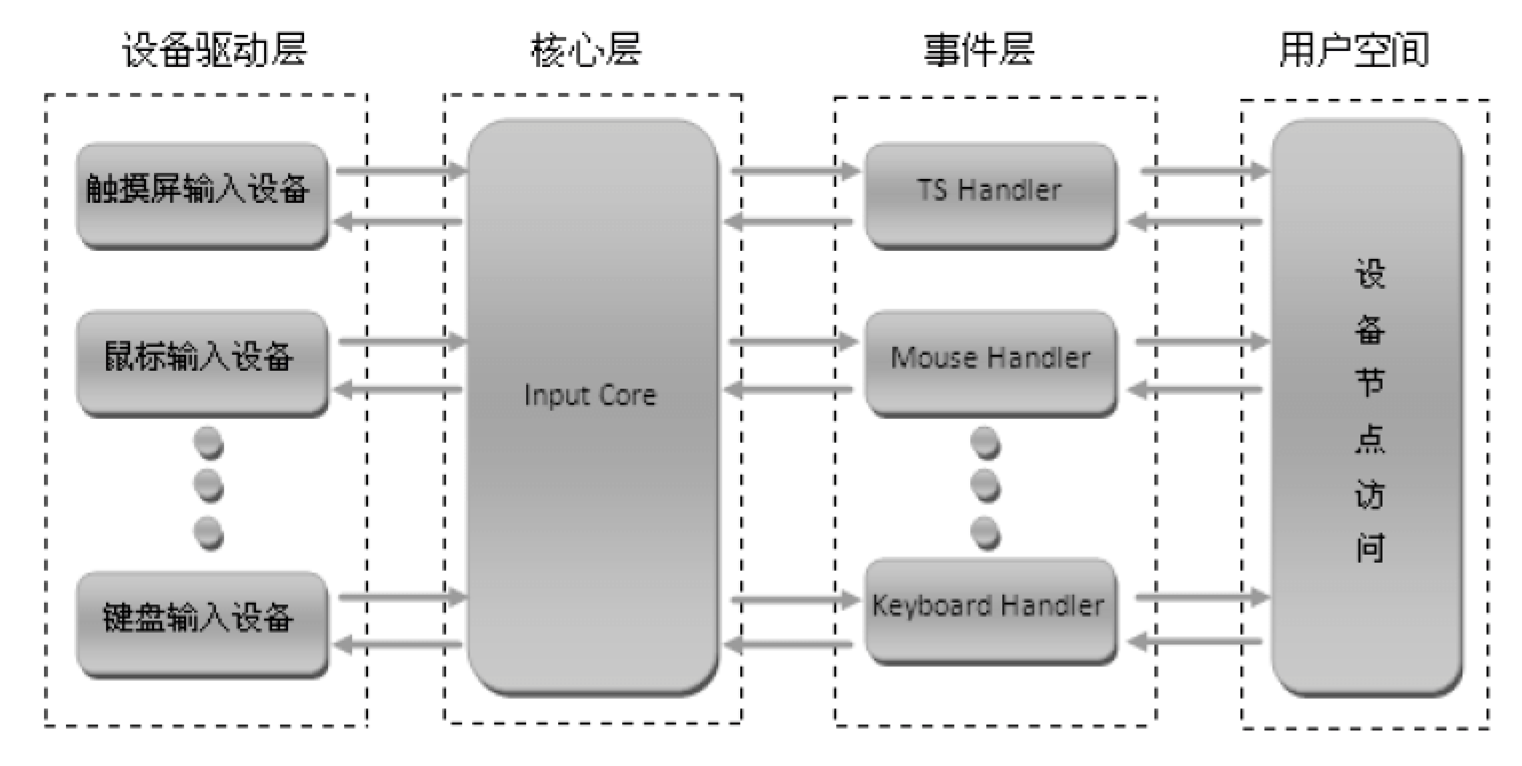
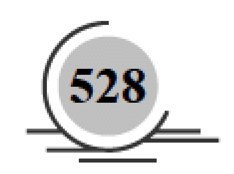


图 17-3　Input 输入子系统的架构图

17.2　移植输入系统驱动的方法

在移植 Android 输入系统驱动时，需要完成如下两个工作。



☑　移植输入（Input）驱动程序。
☑　在用户空间中动态配置 kl 和 kcm 文件。

因为 Android 输入系统的硬件抽象层是 libui 库中的 EventHub，此部分是系统的标准部分。所以在实现特定硬件平台的 Android 系统时，通常需要改变输入系统硬件抽象层。EventHub 使用 Linux 标准的输入设备作为输入设备，并且大多数使用实用的 Event 设备。基于上述原因，为了实现 Android 系统的输入，必须使用 Linux 标准输入驱动程序作为标准的输入。

由此可见，输入系统的标准化程度较高，在用户空间实现时一般不需要更改代码。唯一的变化是使用不同的 kl 和 kcm 文件，使用按键的布局和按键字符映射关系。

17.3　Input（输入）系统驱动详解

在 Android 系统中，Input（输入）驱动程序是 Linux 输入设备的驱动程序，可以进一步分成游戏杆（joystick）、鼠标（mouse 和 mice）和事件设备（Event queue）3 种驱动程序。其中事件驱动程序是目前通用的驱动程序，可以支持键盘、鼠标、触摸屏等多种输入设备。

Input 驱动程序的主设备号是 13，每一种 Input 设备占用 5 位，因此每种设备包含的个数是 32 个。Event 设备在用户空间大多使用如下 3 种文件系统来操作接口。

☑　Read：用于读取输入信息。
☑　Ioctl：用于获得和设置信息。
☑　Poll：调用可以进行用户空间的阻塞，当内核有按键等中断时，通过在中断中唤醒 poll 的内核实现，这样在用户空间的 poll 调用也可以返回。

Event 设备在文件系统中的设备节点为/dev/input/eventX 目录。主设备号为 13，次设备号按照递增顺序生成，为 64～95，各个具体的设备保存在 misc、touchscreen 和 keyboard 等目录中。

Android 输入设备驱动程序的头文件是 include/linux/input.h，核心文件是 drivers/input/input.c，Event 部分的代码文件是 drivers/input/evdev.c。

17.3.1　分析头文件

（1）看按键数值的定义，因为在 Android 手机系统中使用的键盘（keyboard）和小键盘（kaypad）属于按键设备 EV_KEY，轨迹球属于相对设备 EV_REL，触摸屏属于绝对设备 EV_ABS。在文件 input.h 中定义按键数值的代码如下所示。

```
#define KEY_RESERVED        0
#define KEY_ESC             1
#define KEY_1               2
#define KEY_2               3
#define KEY_3               4
#define KEY_4               5
#define KEY_5               6
#define KEY_6               7
#define KEY_7               8
#define KEY_8               9
#define KEY_9               10
#define KEY_0               11
#define KEY_MINUS           12
```

```
#define KEY_EQUAL           13
#define KEY_BACKSPACE       14
#define KEY_TAB             15
#define KEY_Q               16
#define KEY_W               17
#define KEY_E               18
#define KEY_R               19
#define KEY_T               20
```

（2）定义结构体 input_dev，功能是表示 Input 驱动程序的各种信息，在里面定义并归纳了各种设备的信息，例如按键、相对设备、绝对设备、杂项设备、LED、声音设备，强制反馈设备、开关设备等。结构 struct input_dev 的定义代码如下所示。

```
struct input_dev {
    const char *name;
//设备名称
    const char *phys;
//设备在系统的物理路径
    const char *uniq;
//统一的 ID
    struct input_id id;
//设备 ID
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
//事件
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
//按键
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
//相对设备
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];
//绝对设备
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
//杂项设备
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
//LED
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
//声音设备
    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
//强制反馈设备
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)];
//开关设备
    unsigned int keycodemax;
//按键码的最大值
    unsigned int keycodesize;
//按键码的大小
    void *keycode;
//按键码
    int (*setkeycode)(struct input_dev *dev, int
scancode, int keycode);
    int (*getkeycode)(struct input_dev *dev, int
scancode, int *keycode);
    struct ff_device *ff;
    unsigned int repeat_key;
```

```
	struct timer_list timer;
	int sync;
	int abs[ABS_MAX + 1];
	int rep[REP_MAX + 1];
	unsigned long key[BITS_TO_LONGS(KEY_CNT)];
	unsigned long led[BITS_TO_LONGS(LED_CNT)];
	unsigned long snd[BITS_TO_LONGS(SND_CNT)];
	unsigned long sw[BITS_TO_LONGS(SW_CNT)];
	int absmax[ABS_MAX + 1];
//绝对设备相关内容
	int absmin[ABS_MAX + 1];
	int absfuzz[ABS_MAX + 1];
	int absflat[ABS_MAX + 1];
//设备相关的操作
	int (*open)(struct input_dev *dev);
	void (*close)(struct input_dev *dev);
	int (*flush)(struct input_dev *dev, struct file *file);
	int (*event)(struct input_dev *dev, unsigned int type,
			unsigned int code, int value);
	struct input_handle *grab;
	spinlock_t event_lock;
	struct mutex mutex;
	unsigned int users;
	int going_away;
	struct device dev;
	struct list_head	h_list;
	struct list_head	node;
	unsigned int num_vals;
	unsigned int max_vals;
	struct input_value *vals;
	bool devres_managed;
	};
```

（3）在具体实现 Event 驱动程序时，可以使用接口通过向上通知的方式得到按键的事件。在文件 input.h 中定义实现上述接口的代码如下所示。

```
void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value);
void input_inject_event(struct input_handle *handle, unsigned int type, unsigned int code, int value);

static inline void input_report_key(struct input_dev *dev, unsigned int code, int value)
{
		input_event(dev, EV_KEY, code, !!value);
}

static inline void input_report_rel(struct input_dev *dev, unsigned int code, int value)
{
		input_event(dev, EV_REL, code, value);
}

static inline void input_report_abs(struct input_dev *dev, unsigned int code, int value)
{
		input_event(dev, EV_ABS, code, value);
```

```
}

static inline void input_report_ff_status(struct input_dev *dev, unsigned int code, int value)
{
        input_event(dev, EV_FF_STATUS, code, value);
}

static inline void input_report_switch(struct input_dev *dev, unsigned int code, int value)
{
        input_event(dev, EV_SW, code, !!value);
}

static inline void input_sync(struct input_dev *dev)
{
        input_event(dev, EV_SYN, SYN_REPORT, 0);
}

static inline void input_mt_sync(struct input_dev *dev)
{
        input_event(dev, EV_SYN, SYN_MT_REPORT, 0);
}
```

（4）基于文件 input.c 的原理，下面是笔者编写的 USB 输入驱动程序。

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/poll.h>
#include <linux/input.h>

#define USB_MOUSE    ("/dev/input/mouse0")

struct pollfd   mypoll;
int main(int argc,   char *argv[])
{
    int mouseFd;
    struct input_event buff;

        if ((mouseFd = open(USB_MOUSE, O_RDONLY)) == -1) {
            printf("Failed to open /dev/input/mouse0\n");
            return -1;
        }
        mypoll.fd =   mouseFd;
        mypoll.events = POLLIN;
         while(1)
    {
       if(poll( &mypoll, 1, 10) > 0)
      {
            unsigned char data[4] ={0};
            /*
            data 的数据格式:
            data0:00xx 1xxx    ----低 3 位是按键值---左中右分别为 01 02 04，第 4/5 位分别代表 x、y 移动方向，
右上方 x/y>0，左下方 xy<0
```

```
			data1:取值范围-127~127，代表 x 轴移动偏移量
			data2:取值范围-127~127，代表 y 轴移动偏移量
			*/
			usleep(50000);
//MOUSEDEV_EMUL_PS2 方式每次采样数据为 3 个字节，多读不会出错，只返回成功读取的数据数
			read(mouseFd, data, 4);
			printf("mouse data=%02x%02x%02x%02x\n", data[0],data[1], data[2], data[3]);
		}
	}

	close(mouseFd);
		return 0;
}
```

（5）再看结构体 ff_device，表示强制反馈设备的数据，具体实现代码如下所示。

```
struct ff_device {
	int (*upload)(struct input_dev *dev, struct ff_effect *effect,
				struct ff_effect *old);
	int (*erase)(struct input_dev *dev, int effect_id);

	int (*playback)(struct input_dev *dev, int effect_id, int value);
	void (*set_gain)(struct input_dev *dev, u16 gain);
	void (*set_autocenter)(struct input_dev *dev, u16 magnitude);

	void (*destroy)(struct ff_device *);

	void *private;

	unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];

	struct mutex mutex;

	int max_effects;
	struct ff_effect *effects;
	struct file *effect_owners[];
};
```

17.3.2　分析核心文件 input.c

文件 input.c 是输入系统驱动的核心实现，在此文件中包含了大量的操作接口。下面将详细分析文件 input.c 的具体实现。

（1）看函数 input_init()和 input_exit()，功能是实现 Input 设备的初始化和注销工作，具体实现代码如下所示。

```
static int __init input_init(void)
{
	int err;

	err = class_register(&input_class);
	if (err) {
		pr_err("unable to register input_dev class\n");
```

```
			return err;
		}

		err = input_proc_init();
		if (err)
			goto fail1;

		err = register_chrdev_region(MKDEV(INPUT_MAJOR, 0),
					     INPUT_MAX_CHAR_DEVICES, "input");
		if (err) {
			pr_err("unable to register char major %d", INPUT_MAJOR);
			goto fail2;
		}

		return 0;

fail2: input_proc_exit();
fail1: class_unregister(&input_class);
		return err;
}

static void __exit input_exit(void)
{
		input_proc_exit();
		unregister_chrdev_region(MKDEV(INPUT_MAJOR, 0),
					 INPUT_MAX_CHAR_DEVICES);
		class_unregister(&input_class);
}

subsys_initcall(input_init);
module_exit(input_exit);
```

（2）再看函数 input_allocate_device()，功能是实现 Input 设备的分配工作，具体实现代码如下所示。

```
struct input_dev *input_allocate_device(void)
{
		static atomic_t input_no = ATOMIC_INIT(0);
		struct input_dev *dev;

		dev = kzalloc(sizeof(struct input_dev), GFP_KERNEL);
		if (dev) {
			dev->dev.type = &input_dev_type;
			dev->dev.class = &input_class;
			device_initialize(&dev->dev);
			mutex_init(&dev->mutex);
			spin_lock_init(&dev->event_lock);
			init_timer(&dev->timer);
			INIT_LIST_HEAD(&dev->h_list);
			INIT_LIST_HEAD(&dev->node);

			dev_set_name(&dev->dev, "input%ld",
				     (unsigned long) atomic_inc_return(&input_no) - 1);
```

```

                __module_get(THIS_MODULE);
        }

        return dev;
}
EXPORT_SYMBOL(input_allocate_device);
```

（3）再看函数 input_register_device()，功能是实现 Input 设备的注册工作，具体实现代码如下所示。

```
int input_register_device(struct input_dev *dev)
{
        struct input_devres *devres = NULL;
        struct input_handler *handler;
        unsigned int packet_size;
        const char *path;
        int error;

        if (dev->devres_managed) {
                devres = devres_alloc(devm_input_device_unregister,
                                      sizeof(struct input_devres), GFP_KERNEL);
                if (!devres)
                        return -ENOMEM;

                devres->input = dev;
        }

        /* Every input device generates EV_SYN/SYN_REPORT events */
        __set_bit(EV_SYN, dev->evbit);

        /* KEY_RESERVED is not supposed to be transmitted to userspace */
        __clear_bit(KEY_RESERVED, dev->keybit);

        /* Make sure that bitmasks not mentioned in dev->evbit are clean */
        input_cleanse_bitmasks(dev);

        packet_size = input_estimate_events_per_packet(dev);
        if (dev->hint_events_per_packet < packet_size)
                dev->hint_events_per_packet = packet_size;

        dev->max_vals = max(dev->hint_events_per_packet, packet_size) + 2;
        dev->vals = kcalloc(dev->max_vals, sizeof(*dev->vals), GFP_KERNEL);
        if (!dev->vals) {
                error = -ENOMEM;
                goto err_devres_free;
        }

        /*
         * If delay and period are pre-set by the driver, then autorepeating
         * is handled by the driver itself and we don't do it in input.c
         */
        if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
```

```
			dev->timer.data = (long) dev;
			dev->timer.function = input_repeat_key;
			dev->rep[REP_DELAY] = 250;
			dev->rep[REP_PERIOD] = 33;
		}

		if (!dev->getkeycode)
			dev->getkeycode = input_default_getkeycode;

		if (!dev->setkeycode)
			dev->setkeycode = input_default_setkeycode;

		error = device_add(&dev->dev);
		if (error)
			goto err_free_vals;

		path = kobject_get_path(&dev->dev.kobj, GFP_KERNEL);
		pr_info("%s as %s\n",
			dev->name ? dev->name : "Unspecified device",
			path ? path : "N/A");
		kfree(path);

		error = mutex_lock_interruptible(&input_mutex);
		if (error)
			goto err_device_del;

		list_add_tail(&dev->node, &input_dev_list);

		list_for_each_entry(handler, &input_handler_list, node)
			input_attach_handler(dev, handler);

		input_wakeup_procfs_readers();

		mutex_unlock(&input_mutex);

		if (dev->devres_managed) {
			dev_dbg(dev->dev.parent, "%s: registering %s with devres.\n",
				__func__, dev_name(&dev->dev));
			devres_add(dev->dev.parent, devres);
		}
		return 0;

err_device_del:
		device_del(&dev->dev);
err_free_vals:
		kfree(dev->vals);
		dev->vals = NULL;
err_devres_free:
		devres_free(devres);
		return error;
```

```
}
EXPORT_SYMBOL(input_register_device);
```

（4）再看函数 input_unregister_device()，功能是实现 Input 设备的注销工作，具体实现代码如下所示。

```
void input_unregister_device(struct input_dev *dev)
{
	if (dev->devres_managed) {
		WARN_ON(devres_destroy(dev->dev.parent,
					devm_input_device_unregister,
					devm_input_device_match,
					dev));
		__input_unregister_device(dev);
		/*
		 * We do not do input_put_device() here because it will be done
		 * when 2nd devres fires up.
		 */
	} else {
		__input_unregister_device(dev);
		input_put_device(dev);
	}
}
EXPORT_SYMBOL(input_unregister_device);
```

（5）函数 input_proc_init()的功能是建立 input 子系统在 proc 文件系统中的目录和文件，并注册相应的 fops。函数 input_proc_init()的具体实现代码如下所示。

```
static int __init input_proc_init(void)
{
	struct proc_dir_entry *entry;

	proc_bus_input_dir = proc_mkdir("bus/input", NULL);
	if (!proc_bus_input_dir)
		return -ENOMEM;

	entry = proc_create("devices", 0, proc_bus_input_dir,
				&input_devices_fileops);
	if (!entry)
		goto fail1;

	entry = proc_create("handlers", 0, proc_bus_input_dir,
				&input_handlers_fileops);
	if (!entry)
		goto fail2;

	return 0;

fail2: remove_proc_entry("devices", proc_bus_input_dir);
fail1: remove_proc_entry("bus/input", NULL);
	return -ENOMEM;
}
```

（6）函数 input_register_handler()的功能是注册 handler，具体实现代码如下所示。

```
int input_register_handler(struct input_handler *handler)
{
```

```
        struct input_dev *dev;
        int error;

        error = mutex_lock_interruptible(&input_mutex);
        if (error)
                return error;

        INIT_LIST_HEAD(&handler->h_list);

        list_add_tail(&handler->node, &input_handler_list);

        list_for_each_entry(dev, &input_dev_list, node)
                input_attach_handler(dev, handler);

        input_wakeup_procfs_readers();

        mutex_unlock(&input_mutex);
        return 0;
}
EXPORT_SYMBOL(input_register_handler);
```

（7）函数 input_to_handler()的功能是输入先通过所有过滤器处理后的数据，如果没有被筛选出来，则打开所有的句柄，通过 dev->event_loc 调用实现中断禁止操作。函数 input_to_handler()的具体实现代码如下所示。

```
static unsigned int input_to_handler(struct input_handle *handle,
                        struct input_value *vals, unsigned int count)
{
        struct input_handler *handler = handle->handler;
        struct input_value *end = vals;
        struct input_value *v;

        for (v = vals; v != vals + count; v++) {
                if (handler->filter &&
                    handler->filter(handle, v->type, v->code, v->value))
                        continue;
                if (end != v)
                        *end = *v;
                end++;
        }

        count = end - vals;
        if (!count)
                return 0;

        if (handler->events)
                handler->events(handle, vals, count);
        else if (handler->event)
                for (v = vals; v != end; v++)
                        handler->event(handle, v->type, v->code, v->value);

```

```
        return count;
}
```

（8）函数 input_handle_event()的功能是判断输入的 type 类型是否支持，并接着进入处理核心。函数 input_handle_event()的具体实现代码如下所示。

```
static void input_handle_event(struct input_dev *dev,
                              unsigned int type, unsigned int code, int value)
{
        int disposition;

        disposition = input_get_disposition(dev, type, code, value);

        if ((disposition & INPUT_PASS_TO_DEVICE) && dev->event)
                dev->event(dev, type, code, value);

        if (!dev->vals)
                return;

        if (disposition & INPUT_PASS_TO_HANDLERS) {
                struct input_value *v;

                if (disposition & INPUT_SLOT) {
                        v = &dev->vals[dev->num_vals++];
                        v->type = EV_ABS;
                        v->code = ABS_MT_SLOT;
                        v->value = dev->mt->slot;
                }

                v = &dev->vals[dev->num_vals++];
                v->type = type;
                v->code = code;
                v->value = value;
        }

        if (disposition & INPUT_FLUSH) {
                if (dev->num_vals >= 2)
                        input_pass_values(dev, dev->vals, dev->num_vals);
                dev->num_vals = 0;
        } else if (dev->num_vals >= dev->max_vals - 2) {
                dev->vals[dev->num_vals++] = input_value_sync;
                input_pass_values(dev, dev->vals, dev->num_vals);
                dev->num_vals = 0;
        }

}
```

（9）函数 input_get_disposition()的功能是获得事件处理者身份。INPUT_PASS_TO_HANDLERS 表示交给 input hardler 处理，INPUT_PASS_TO_DEVICE 表示交给 input device 处理，INPUT_FLUSH 表示需要 handler 立即处理。如果事件正常，一般返回的是 INPUT_PASS_TO_HANDLERS，只有 code 为 SYN_REPORT 时才会返回 INPUT_PASS_TO_HANDLERS | INPUT_FLUSH。函数 input_get_disposition()的具体实现代码如下所示。

```
static int input_get_disposition(struct input_dev *dev,
                    unsigned int type, unsigned int code, int value)
{
        int disposition = INPUT_IGNORE_EVENT;

        switch (type) {

        case EV_SYN:
                switch (code) {
                case SYN_CONFIG:
                        disposition = INPUT_PASS_TO_ALL;
                        break;

                case SYN_REPORT:
                        disposition = INPUT_PASS_TO_HANDLERS | INPUT_FLUSH;
                        break;
                case SYN_MT_REPORT:
                        disposition = INPUT_PASS_TO_HANDLERS;
                        break;
                }
                break;

        case EV_KEY:
                if (is_event_supported(code, dev->keybit, KEY_MAX)) {

                        /* auto-repeat bypasses state updates */
                        if (value == 2) {
                                disposition = INPUT_PASS_TO_HANDLERS;
                                break;
                        }

                        if (!!test_bit(code, dev->key) != !!value) {

                                __change_bit(code, dev->key);
                                disposition = INPUT_PASS_TO_HANDLERS;
                        }
                }
                break;

        case EV_SW:
                if (is_event_supported(code, dev->swbit, SW_MAX) &&
                    !!test_bit(code, dev->sw) != !!value) {

                        __change_bit(code, dev->sw);
                        disposition = INPUT_PASS_TO_HANDLERS;
                }
                break;

        case EV_ABS:
                if (is_event_supported(code, dev->absbit, ABS_MAX))
                        disposition = input_handle_abs_event(dev, code, &value);
```

```

		break;

	case EV_REL:
		if (is_event_supported(code, dev->relbit, REL_MAX) && value)
			disposition = INPUT_PASS_TO_HANDLERS;

		break;

	case EV_MSC:
		if (is_event_supported(code, dev->mscbit, MSC_MAX))
			disposition = INPUT_PASS_TO_ALL;

		break;

	case EV_LED:
		if (is_event_supported(code, dev->ledbit, LED_MAX) &&
		    !!test_bit(code, dev->led) != !!value) {

			__change_bit(code, dev->led);
			disposition = INPUT_PASS_TO_ALL;
		}
		break;

	case EV_SND:
		if (is_event_supported(code, dev->sndbit, SND_MAX)) {

			if (!!test_bit(code, dev->snd) != !!value)
				__change_bit(code, dev->snd);
			disposition = INPUT_PASS_TO_ALL;
		}
		break;

	case EV_REP:
		if (code <= REP_MAX && value >= 0 && dev->rep[code] != value) {
			dev->rep[code] = value;
			disposition = INPUT_PASS_TO_ALL;
		}
		break;

	case EV_FF:
		if (value >= 0)
			disposition = INPUT_PASS_TO_ALL;
		break;

	case EV_PWR:
		disposition = INPUT_PASS_TO_ALL;
		break;
	}

```

```
	return disposition;
}
```

（10）函数 input_handle_abs_event()的功能是过滤掉上次值和这次值相同的事件。函数 input_handle_abs_event()的具体实现代码如下所示。

```
static int input_handle_abs_event(struct input_dev *dev,
				unsigned int code, int *pval)
{
	struct input_mt *mt = dev->mt;
	bool is_mt_event;
	int *pold;

	if (code == ABS_MT_SLOT) {
		/*
		 * "Stage" the event; we'll flush it later, when we
		 * get actual touch data.
		 */
		if (mt && *pval >= 0 && *pval < mt->num_slots)
			mt->slot = *pval;

		return INPUT_IGNORE_EVENT;
	}

	is_mt_event = input_is_mt_value(code);

	if (!is_mt_event) {
		pold = &dev->absinfo[code].value;
	} else if (mt) {
		pold = &mt->slots[mt->slot].abs[code - ABS_MT_FIRST];
	} else {
		/*
		 * Bypass filtering for multi-touch events when
		 * not employing slots.
		 */
		pold = NULL;
	}

	if (pold) {
		*pval = input_defuzz_abs_event(*pval, *pold,
					dev->absinfo[code].fuzz);
		if (*pold == *pval)
			return INPUT_IGNORE_EVENT;

		*pold = *pval;
	}

	/* Flush pending "slot" event */
	if (is_mt_event && mt && mt->slot != input_abs_get_val(dev, ABS_MT_SLOT)) {
		input_abs_set_val(dev, ABS_MT_SLOT, mt->slot);
		return INPUT_PASS_TO_HANDLERS | INPUT_SLOT;
	}
```

```

        return INPUT_PASS_TO_HANDLERS;
}
```

在上述过滤处理过程中，如果 code 不是在 ABS_MT_FIRST 到 ABS_MT_LAST 之间，那就是单点上报（如 ABS_X），否则符合多点上报。上述各种情况的事件值 value 存储的位置是不一样的，所以取 pold 指针的方式也不同（这个 pold 是过滤之后存储的*pold = *pval;）。input_defuzz_abs_event()会对比当前 value 和上一次的 old value，如果一样就过滤掉而不会产生任何事件。

（11）函数 input_pass_values()的功能是处理通过过滤的输入值，具体实现代码如下所示。

```
static void input_pass_values(struct input_dev *dev,
                                      struct input_value *vals, unsigned int count)
{
        struct input_handle *handle;
        struct input_value *v;

        if (!count)
                return;

        rcu_read_lock();

        handle = rcu_dereference(dev->grab);
        if (handle) {
                count = input_to_handler(handle, vals, count);
        } else {
                list_for_each_entry_rcu(handle, &dev->h_list, d_node)
                        if (handle->open)
                                count = input_to_handler(handle, vals, count);
        }

        rcu_read_unlock();

        add_input_randomness(vals->type, vals->code, vals->value);

        /* trigger auto repeat for key events */
        for (v = vals; v != vals + count; v++) {
                if (v->type == EV_KEY && v->value != 2) {
                        if (v->value)
                                input_start_autorepeat(dev, v->code);
                        else
                                input_stop_autorepeat(dev);
                }
        }
}
```

（12）函数 input_inject_event()的功能是向底层发送事件，具体实现代码如下所示。

```
void input_inject_event(struct input_handle *handle,
                                unsigned int type, unsigned int code, int value)
{
        struct input_dev *dev = handle->dev;
        struct input_handle *grab;
        unsigned long flags;
```

```

        if (is_event_supported(type, dev->evbit, EV_MAX)) {
                spin_lock_irqsave(&dev->event_lock, flags);

                rcu_read_lock();
                grab = rcu_dereference(dev->grab);
                if (!grab || grab == handle)
                        input_handle_event(dev, type, code, value);
                rcu_read_unlock();

                spin_unlock_irqrestore(&dev->event_lock, flags);
        }
}
EXPORT_SYMBOL(input_inject_event);
```

（13）函数 input_grab_device()是一个 grap 抓取处理句柄函数，当输入希望处理自己的设备时，输入到处理设备中所产生的所有事件都传递这个句柄。函数 input_grab_device()的具体实现代码如下所示。

```
int input_grab_device(struct input_handle *handle)
{
        struct input_dev *dev = handle->dev;
        int retval;

        retval = mutex_lock_interruptible(&dev->mutex);
        if (retval)
                return retval;

        if (dev->grab) {
                retval = -EBUSY;
                goto out;
        }

        rcu_assign_pointer(dev->grab, handle);

out:
        mutex_unlock(&dev->mutex);
        return retval;
}
EXPORT_SYMBOL(input_grab_device);
```

（14）再看函数 input_release_device()，当一个进程 grabbed 一个设备后进行释放处理时调用。函数 input_release_device()的具体实现代码如下所示。

```
void input_release_device(struct input_handle *handle)
{
        struct input_dev *dev = handle->dev;

        mutex_lock(&dev->mutex);
        __input_release_device(handle);
        mutex_unlock(&dev->mutex);
}
EXPORT_SYMBOL(input_release_device);
```

（15）函数 input_open_device()的功能是打开输入设备，如果 open 成功会更新 evdev->open 计数。函数 input_open_device()的具体实现代码如下所示。



```
int input_open_device(struct input_handle *handle)
{
	struct input_dev *dev = handle->dev;
	int retval;

	retval = mutex_lock_interruptible(&dev->mutex);
	if (retval)
		return retval;

	if (dev->going_away) {
		retval = -ENODEV;
		goto out;
	}

	handle->open++;

	if (!dev->users++ && dev->open)
		retval = dev->open(dev);

	if (retval) {
		dev->users--;
		if (!--handle->open) {
			/*
			 * Make sure we are not delivering any more events
			 * through this handle
			 */
			synchronize_rcu();
		}
	}

out:
	mutex_unlock(&dev->mutex);
	return retval;
}
EXPORT_SYMBOL(input_open_device);
```

（16）函数 input_flush_device()的功能是“冲洗”处理输入设备，具体实现代码如下所示。

```
int input_flush_device(struct input_handle *handle, struct file *file)
{
	struct input_dev *dev = handle->dev;
	int retval;

	retval = mutex_lock_interruptible(&dev->mutex);
	if (retval)
		return retval;

	if (dev->flush)
		retval = dev->flush(dev, file);

	mutex_unlock(&dev->mutex);
	return retval;
```

```
}
EXPORT_SYMBOL(input_flush_device);
```

（17）函数 input_default_setkeycode()的功能是，如果没有定义有关重复按键的相关值，则用内核默认的按键值。函数 input_default_setkeycode()的具体实现代码如下所示。

```
static int input_default_setkeycode(struct input_dev *dev,
                                    const struct input_keymap_entry *ke,
                                    unsigned int *old_keycode)
{
        unsigned int index;
        int error;
        int i;

        if (!dev->keycodesize)
                return -EINVAL;

        if (ke->flags & INPUT_KEYMAP_BY_INDEX) {
                index = ke->index;
        } else {
                error = input_scancode_to_scalar(ke, &index);
                if (error)
                        return error;
        }

        if (index >= dev->keycodemax)
                return -EINVAL;

        if (dev->keycodesize < sizeof(ke->keycode) &&
                        (ke->keycode >> (dev->keycodesize * 8)))
                return -EINVAL;

        switch (dev->keycodesize) {
                case 1: {
                        u8 *k = (u8 *)dev->keycode;
                        *old_keycode = k[index];
                        k[index] = ke->keycode;
                        break;
                }
                case 2: {
                        u16 *k = (u16 *)dev->keycode;
                        *old_keycode = k[index];
                        k[index] = ke->keycode;
                        break;
                }
                default: {
                        u32 *k = (u32 *)dev->keycode;
                        *old_keycode = k[index];
                        k[index] = ke->keycode;
                        break;
                }
        }
```

```

        __clear_bit(*old_keycode, dev->keybit);
        __set_bit(ke->keycode, dev->keybit);

        for (i = 0; i < dev->keycodemax; i++) {
                if (input_fetch_keycode(dev, i) == *old_keycode) {
                        __set_bit(*old_keycode, dev->keybit);
                        break; /* Setting the bit twice is useless, so break */
                }
        }

        return 0;
}
```

（18）函数 input_devices_seq_show()的功能是打印输出信息到 seq 文件，通过 cat 命令调用 read 方法，在 read 方法中调用 show 方法来展示输入信息。函数 input_devices_seq_show()的具体实现代码如下所示。

```
static int input_devices_seq_show(struct seq_file *seq, void *v)
{
        struct input_dev *dev = container_of(v, struct input_dev, node);
        const char *path = kobject_get_path(&dev->dev.kobj, GFP_KERNEL);
        struct input_handle *handle;

        seq_printf(seq, "I: Bus=%04x Vendor=%04x Product=%04x Version=%04x\n",
                   dev->id.bustype, dev->id.vendor, dev->id.product, dev->id.version);

        seq_printf(seq, "N: Name=\"%s\"\n", dev->name ? dev->name : "");
        seq_printf(seq, "P: Phys=%s\n", dev->phys ? dev->phys : "");
        seq_printf(seq, "S: Sysfs=%s\n", path ? path : "");
        seq_printf(seq, "U: Uniq=%s\n", dev->uniq ? dev->uniq : "");
        seq_printf(seq, "H: Handlers=");

        list_for_each_entry(handle, &dev->h_list, d_node)
                seq_printf(seq, "%s ", handle->name);
        seq_putc(seq, '\n');

        input_seq_print_bitmap(seq, "PROP", dev->propbit, INPUT_PROP_MAX);

        input_seq_print_bitmap(seq, "EV", dev->evbit, EV_MAX);
        if (test_bit(EV_KEY, dev->evbit))
                input_seq_print_bitmap(seq, "KEY", dev->keybit, KEY_MAX);
        if (test_bit(EV_REL, dev->evbit))
                input_seq_print_bitmap(seq, "REL", dev->relbit, REL_MAX);
        if (test_bit(EV_ABS, dev->evbit))
                input_seq_print_bitmap(seq, "ABS", dev->absbit, ABS_MAX);
        if (test_bit(EV_MSC, dev->evbit))
                input_seq_print_bitmap(seq, "MSC", dev->mscbit, MSC_MAX);
        if (test_bit(EV_LED, dev->evbit))
                input_seq_print_bitmap(seq, "LED", dev->ledbit, LED_MAX);
        if (test_bit(EV_SND, dev->evbit))
                input_seq_print_bitmap(seq, "SND", dev->sndbit, SND_MAX);
        if (test_bit(EV_FF, dev->evbit))
```

```
                input_seq_print_bitmap(seq, "FF", dev->ffbit, FF_MAX);
        if (test_bit(EV_SW, dev->evbit))
                input_seq_print_bitmap(seq, "SW", dev->swbit, SW_MAX);

        seq_putc(seq, '\n');

        kfree(path);
        return 0;
}
```

（19）函数 input_reset_device()的功能是把一个 input_dev 添加到 input_dev_list 链表上，并同时在链表 input_handler_list 中找到和这个 input_dev 相匹配的结构 input_handler，并把相匹配的 input_dev 和 input_handler 连接（connect）起来（通过 input_handle 建立连接关系）。当连接成功之后，在 input_dev 上发生的中断事件就可以传递到 input_handler 中，从而可以进一步传递到用户空间中。函数 input_reset_device 的具体实现代码如下所示。

```
void input_reset_device(struct input_dev *dev)
{
        mutex_lock(&dev->mutex);

        if (dev->users) {
                input_dev_toggle(dev, true);

                /*
                 * Keys that have been pressed at suspend time are unlikely
                 * to be still pressed when we resume.
                 */
                spin_lock_irq(&dev->event_lock);
                input_dev_release_keys(dev);
                spin_unlock_irq(&dev->event_lock);
        }

        mutex_unlock(&dev->mutex);
}
EXPORT_SYMBOL(input_reset_device);
```

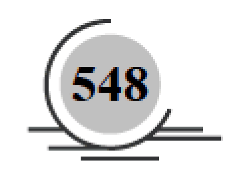
通过本节内容的介绍，可以总结出输入系统驱动事件的传递过程：首先在驱动层调用 inport_report_abs，然后调用 input core 层的 input_event()，input_event()调用了 input_handle_event 对事件进行分派，调用 input_pass_event()，在这里它会把事件传递给具体的 handler 层，然后在相应 handler 的 event 处理函数中封装一个 event，然后把它投入到 evdev 的 client_list 列表中的客户端事件 buffer 中，等待用户空间来读取。

17.3.3 event 机制详解

在 Android 系统中，输入系统 event 机制的实现文件是 driver/input/event.c，下面将详细分析 event 机制的具体实现过程。

（1）在 Linux 内核系统中，使用结构体 input_dev 来描述一个 Input 设备，该结构的定义代码如下所示。

```
struct input_dev {
    struct input_id id;/*指向 input_id 结构*/
    bool sync;
    struct device dev;/**这些设备都归属总线设备模型*/
```



```
    struct list_head        h_list;
    struct list_head        node;   //input_handle 链表的 list 节点
};
```

在内核中使用 input_register_device(struct input_dev *dev)来注册一个 Input 设备，用 input_handler 表示 Input 设备的接口，使用 input_register_handler(structinput_handler *handler)实现注册功能。

（2）在 Event 事件驱动实现过程中，实现 Input 设备注册的代码如下所示。

```
int input_register_device(struct input_dev *dev)
{
    static atomic_t input_no = ATOMIC_INIT(0);
    struct input_handler *handler;
    const char *path;
    int error;
    /* Every input device generates EV_SYN/SYN_REPORT events */
    __set_bit(EV_SYN, dev->evbit);//see to inpu.h
    /* KEY_RESERVED is not supposed to be transmitted to userspace */
    __clear_bit(KEY_RESERVED, dev->keybit);
    /* Make sure that bitmasks not mentioned in dev->evbit are clean */
    input_cleanse_bitmasks(dev);
    /*
     * If delay and period are pre-set by the driver, then autorepeating
     * is handled by the driver itself and we don't do it in input.c.
     */
    init_timer(&dev->timer);
    //处理重复按键，如果没赋值则为其赋默认的值
    if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
        dev->timer.data = (long) dev;
        dev->timer.function = input_repeat_key;
        dev->rep[REP_DELAY] = 250;
        dev->rep[REP_PERIOD] = 33;
    }
    if (!dev->getkeycode)//获取键的扫描码
        dev->getkeycode = input_default_getkeycode;
    if (!dev->setkeycode)//设置键值
        dev->setkeycode = input_default_setkeycode;
    dev_set_name(&dev->dev, "input%ld",
             (unsigned long) atomic_inc_return(&input_no) - 1);
    //将 input_dev 中封装的 device 注册到 sysfs
    error = device_add(&dev->dev);
    if (error)
        return error;
    path = kobject_get_path(&dev->dev.kobj, GFP_KERNEL);
    printk(KERN_INFO "input: %s as %s\n",
        dev->name ? dev->name : "Unspecified device", path ? path : "N/A");
    kfree(path);
    error = mutex_lock_interruptible(&input_mutex);
    if (error) {
        device_del(&dev->dev);
        return error;
    }
    //将 input_device 挂到 input_dev_list 链表中
```

```
    list_add_tail(&dev->node, &input_dev_list);
    //对挂载在 input_dev_list 中的每一个 handler 调用 input_attach_handler(dev, handler)
    list_for_each_entry(handler, &input_handler_list, node)
        input_attach_handler(dev, handler);
    input_wakeup_procfs_readers();
    mutex_unlock(&input_mutex);
    return 0;
}
```

在上述代码中，首先将 input_device 挂载到 input_dev_list 链表上，然后对挂载在 input_dev_list 中的每一个 handler 调用 input_attach_handler(dev, handler)进行匹配。例如，设备模型中的 device 和 driver 的匹配，所有的 input device 都挂载在 input_dev_list 上，所有的 handler 都挂载在 input_handler_list 上。

（3）再看函数 input_attach_handler()，功能是调用函数 input_match_device()对 handler 和 dev 通过 input_device_id *id 进行匹配操作。如果匹配成功，则调用 handler->connect 来关联结构 input_dev *dev 和结构 input_handler *handler。函数 input_attach_handler()的具体实现代码如下所示。

```
static int input_attach_handler(struct input_dev *dev, struct input_handler *handler)
{
    const struct input_device_id *id;
    int error;
    id = input_match_device(handler, dev);
    if (!id)
        return -ENODEV;
    error = handler->connect(handler, dev, id);
    if (error && error != -ENODEV)
        printk(KERN_ERR
            "input: failed to attach handler %s to device %s, "
            "error: %d\n",
            handler->name, kobject_name(&dev->dev.kobj), error);
    return error;
}
```

函数 input_match_device()的具体实现代码如下所示。

```
static const struct input_device_id *input_match_device(struct input_handler *handler,struct input_dev *dev)
{
    const struct input_device_id *id;
    int i;
    for (id = handler->id_table; id->flags || id->driver_info; id++) {//flags 配置匹配的类型
        if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)//匹配总线类型
            if (id->bustype != dev->id.bustype)
                continue;
        if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)//匹配厂商
            if (id->vendor != dev->id.vendor)
                continue;
        if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)//匹配制造商
            if (id->product != dev->id.product)
                continue;
        if (id->flags & INPUT_DEVICE_ID_MATCH_VERSION)//匹配版本号
            if (id->version != dev->id.version)
                continue;
      //如果上面的 id->flags 匹配成功或者是 id->flags 没有定义则执行下面的函数
        MATCH_BIT(evbit,  EV_MAX);
```

```
            MATCH_BIT(keybit, KEY_MAX);
            MATCH_BIT(relbit, REL_MAX);
            MATCH_BIT(absbit, ABS_MAX);
            MATCH_BIT(mscbit, MSC_MAX);
            MATCH_BIT(ledbit, LED_MAX);
            MATCH_BIT(sndbit, SND_MAX);
            MATCH_BIT(ffbit,   FF_MAX);
            MATCH_BIT(swbit,   SW_MAX);
            if (!handler->match || handler->match(handler, dev))
                return id;
        }
        return NULL;
}
```

17.4　硬件抽象层详解

通过 17.3 节内容的讲解，读者已经基本了解了 Android 系统中输入系统驱动的实现内核。本节将详细讲解硬件抽象层的实现过程。

17.4.1　处理用户空间

在 Android 系统中，文件 frameworks/base/include/androidfw/KeycodeLabels.h 是本地框架层 libui 的头文件，用于实现用户空间处理功能。现实中的触摸屏和轨迹球通常非常简单，只需要传递坐标、按下、抬起等信息即可。而按键处理的过程稍微复杂，按键表示方式需要先后经过按键布局转换和按键码转换。

注意：键扫描码 Scancode 是由 Linux 的输入驱动框架定义的整数类型。键扫描码 Scancode 经过一次转化后，形成按键标签 KeycodeLabel，这是一个字符串的表示形式。按键标签 KeycodeLabel 经过转换后，再次形成整数型按键码 keycode。在 Android 应用程序层，主要使用按键码 Keycode 来区分。

（1）在文件 KeycodeLabels.h 中，按键码是整数值的格式，在此文件中是使用枚举实现的，枚举 KeyCode 的定义代码如下所示。

```
typedef enum KeyCode {
    kKeyCodeUnknown = 0,
    kKeyCodeSoftLeft = 1,
    kKeyCodeSoftRight = 2,
    kKeyCodeHome = 3,
    kKeyCodeBack = 4,
    kKeyCodeCall = 5,
    kKeyCodeEndCall = 6,
    kKeyCode0 = 7,
    kKeyCode1 = 8,
    kKeyCode2 = 9,
    kKeyCode3 = 10,
    kKeyCode4 = 11,
    kKeyCode5 = 12,
    kKeyCode6 = 13,
    kKeyCode7 = 14,
```

```
    kKeyCode8 = 15,
    kKeyCode9 = 16,
    kKeyCodeStar = 17,
    kKeyCodePound = 18,
    kKeyCodeDpadUp = 19,
    kKeyCodeDpadDown = 20,
    kKeyCodeDpadLeft = 21,
    kKeyCodeDpadRight = 22,
    kKeyCodeDpadCenter = 23,
    kKeyCodeVolumeUp = 24,
    kKeyCodeVolumeDown = 25,
    kKeyCodePower = 26,
    kKeyCodeCamera = 27,
    kKeyCodeClear = 28,
    kKeyCodeA = 29,
    kKeyCodeB = 30,
    kKeyCodeC = 31,
    kKeyCodeD = 32,
    kKeyCodeE = 33,
    kKeyCodeF = 34,
    kKeyCodeG = 35,
    kKeyCodeH = 36,
    kKeyCodeI = 37,
    kKeyCodeJ = 38,
    kKeyCodeK = 39,
    kKeyCodeL = 40,
    kKeyCodeM = 41,
    kKeyCodeN = 42,
    kKeyCodeO = 43,
    kKeyCodeP = 44,
    kKeyCodeQ = 45,
    kKeyCodeR = 46,
    kKeyCodeS = 47,
    kKeyCodeT = 48,
    kKeyCodeU = 49,
    kKeyCodeV = 50,
    kKeyCodeW = 51,
    kKeyCodeX = 52,
    kKeyCodeY = 53,
    kKeyCodeZ = 54,
    kKeyCodeComma = 55,
    kKeyCodePeriod = 56,
    kKeyCodeAltLeft = 57,
    kKeyCodeAltRight = 58,
    kKeyCodeShiftLeft = 59,
    kKeyCodeShiftRight = 60,
    kKeyCodeTab = 61,
    kKeyCodeSpace = 62,
    kKeyCodeSym = 63,
    kKeyCodeExplorer = 64,
    kKeyCodeEnvelope = 65,
```

```
    kKeyCodeNewline = 66,
    kKeyCodeDel = 67,
    kKeyCodeGrave = 68,
    kKeyCodeMinus = 69,
    kKeyCodeEquals = 70,
    kKeyCodeLeftBracket = 71,
    kKeyCodeRightBracket = 72,
    kKeyCodeBackslash = 73,
    kKeyCodeSemicolon = 74,
    kKeyCodeApostrophe = 75,
    kKeyCodeSlash = 76,
    kKeyCodeAt = 77,
    kKeyCodeNum = 78,
    kKeyCodeHeadSetHook = 79,
    kKeyCodeFocus = 80,
    kKeyCodePlus = 81,
    kKeyCodeMenu = 82,
    kKeyCodeNotification = 83,
    kKeyCodeSearch = 84,
    kKeyCodePlayPause = 85,
    kKeyCodeStop = 86,
    kKeyCodeNextSong = 87,
    kKeyCodePreviousSong = 88,
    kKeyCodeRewind = 89,
    kKeyCodeForward = 90,
    kKeyCodeMute = 91
} KeyCode;
```

（2）定义数组 KEYCODES[]，功能是存储从字符串到整数的映射关系。左列的内容表示按键标签 KeyCodeLabel，右列的内容表示按键码 KeyCode（与 KeyCode 的数值对应）。其实在按键信息第二次转化时，是将字符串类型 KeyCodeLabel 转化成了整数的 KeyCode。定义数组 KEYCODES[]的代码如下所示。

```
static const KeycodeLabel KEYCODES[] = {
    { "SOFT_LEFT", 1 },
    { "SOFT_RIGHT", 2 },
    { "HOME", 3 },
    { "BACK", 4 },
    { "CALL", 5 },
    { "ENDCALL", 6 },
    { "0", 7 },
    { "1", 8 },
    { "2", 9 },
    { "3", 10 },
    { "4", 11 },
    { "5", 12 },
    { "6", 13 },
    { "7", 14 },
    { "8", 15 },
    { "9", 16 },
    { "STAR", 17 },
    { "POUND", 18 },
    { "DPAD_UP", 19 },
```

```
{ "DPAD_DOWN", 20 },
{ "DPAD_LEFT", 21 },
{ "DPAD_RIGHT", 22 },
{ "DPAD_CENTER", 23 },
{ "VOLUME_UP", 24 },
{ "VOLUME_DOWN", 25 },
{ "POWER", 26 },
{ "CAMERA", 27 },
{ "CLEAR", 28 },
{ "A", 29 },
{ "B", 30 },
{ "C", 31 },
{ "D", 32 },
{ "E", 33 },
{ "F", 34 },
{ "G", 35 },
{ "H", 36 },
{ "I", 37 },
{ "J", 38 },
{ "K", 39 },
{ "L", 40 },
{ "M", 41 },
{ "N", 42 },
{ "O", 43 },
{ "P", 44 },
{ "Q", 45 },
{ "R", 46 },
{ "S", 47 },
{ "T", 48 },
{ "U", 49 },
{ "V", 50 },
{ "W", 51 },
{ "X", 52 },
{ "Y", 53 },
{ "Z", 54 },
{ "COMMA", 55 },
{ "PERIOD", 56 },
{ "ALT_LEFT", 57 },
{ "ALT_RIGHT", 58 },
{ "SHIFT_LEFT", 59 },
{ "SHIFT_RIGHT", 60 },
{ "TAB", 61 },
{ "SPACE", 62 },
{ "SYM", 63 },
{ "EXPLORER", 64 },
{ "ENVELOPE", 65 },
{ "ENTER", 66 },
{ "DEL", 67 },
{ "GRAVE", 68 },
{ "MINUS", 69 },
```

```
    { "EQUALS", 70 },
    { "LEFT_BRACKET", 71 },
    { "RIGHT_BRACKET", 72 },
    { "BACKSLASH", 73 },
    { "SEMICOLON", 74 },
    { "APOSTROPHE", 75 },
    { "SLASH", 76 },
    { "AT", 77 },
    { "NUM", 78 },
    { "HEADSETHOOK", 79 },
    { "FOCUS", 80 },
    { "PLUS", 81 },
    { "MENU", 82 },
    { "NOTIFICATION", 83 },
    { "SEARCH", 84 },
    { "MEDIA_PLAY_PAUSE", 85 },
    { "MEDIA_STOP", 86 },
    { "MEDIA_NEXT", 87 },
    { "MEDIA_PREVIOUS", 88 },
    { "MEDIA_REWIND", 89 },
    { "MEDIA_FAST_FORWARD", 90 },
    { "MUTE", 91 },
    { NULL, 0 }
};
```

注意： 在文件 frameworks/base/core/Java/android/view/KeyEvent.Java 中定义了 android.view.KeyEvent 类，在里面定义整数类型的数值与 KeycodeLabels.h 中定义的枚举 KeyCode 值是对应的。

17.4.2　定义按键的字符映射关系

在 Android 系统中，文件 frameworks/base/include/androidfw/KeyCharacterMap.h 也是本地框架层 libui 的头文件，在里面定义了按键的字符映射关系。其实 KeyCharacterMap 只是一个辅助的功能，因为按键码只是一个与 UI 无关的整数，通常用程序对其进行捕获处理，然而如果将按键事件转换为用户可见的内容，就需要经过这个层次的转换。其中定义类 KeyCharacterMap 的实现代码如下所示。

```
class KeyCharacterMap : public RefBase {
public:
    enum KeyboardType {
        KEYBOARD_TYPE_UNKNOWN = 0,
        KEYBOARD_TYPE_NUMERIC = 1,
        KEYBOARD_TYPE_PREDICTIVE = 2,
        KEYBOARD_TYPE_ALPHA = 3,
        KEYBOARD_TYPE_FULL = 4,
        KEYBOARD_TYPE_SPECIAL_FUNCTION = 5,
        KEYBOARD_TYPE_OVERLAY = 6,
    };

    enum Format {
        // Base keyboard layout, may contain device-specific options, such as "type" declaration.
        FORMAT_BASE = 0,
```

```
            // Overlay keyboard layout, more restrictive, may be published by applications,
            // cannot override device-specific options.
            FORMAT_OVERLAY = 1,
            // Either base or overlay layout ok.
            FORMAT_ANY = 2,
        };

        // Substitute key code and meta state for fallback action.
        struct FallbackAction {
            int32_t keyCode;
            int32_t metaState;
        };
```

在上述代码中，使用 KeyCharacterMap 将按键码映射为文本可识别的字符串。上述关于按键码和按键字符映射的内容是在代码中实现的内容，我们还需要配合动态的配置文件来使用。在实现 Android 系统时，很可能需要更改这两种文件。我们需要动态配置如下两个文件。

☑ KL（Keycode Layout）：后缀名为 kl 的配置文件。

☑ KCM（KeyCharacterMap）：后缀名为 kcm 的配置文件。

在 Donut 及其之前版本的配置文件路径为 development/emulator/keymaps/。

在 Eclair 及其之后版本的配置文件路径为 sdk/emulator/keymaps/。

当系统生成上述配置文件后，会将其放置在目标文件系统的/system/usr/keylayout/目录中或/system/usr/keychars/目录中。另外，KL 文件将被直接复制到目标文件系统中；由于尺寸较大，KCM 文件放置在目标文件系统中之前，需要经过压缩处理。KeyLayoutMap.cpp 负责解析处理 kl 文件，KeyCharacterMap.cpp 负责解析 kcm 文件。

17.4.3 KL 格式的按键布局文件

在 Android 系统中，KL 格式文件是按键布局文件，通常以原始的文本文件形式存在，被保存在目标文件系统的/system/usr/keylayout/目录中或者/system/usr/keychars/目录中。Android 默认提供的按键布局文件有两个，分别是 qwerty.kl 和 AVRCP.kl。其中 qwerty.kl 是全键盘的布局文件，是系统中主要按键使用的布局文件；文件 AVRCP.kl 用于实现多媒体的控制。

文件 qwerty.kl 的主要内容如下所示。

```
key 399    GRAVE
key 2      1
key 3      2
key 4      3
key 5      4
key 6      5
key 7      6
key 8      7
key 9      8
key 10     9
key 11     0
key 158    BACK                WAKE_DROPPED
key 230    SOFT_RIGHT          WAKE
key 60     SOFT_RIGHT          WAKE
key 107    ENDCALL             WAKE_DROPPED
```

```
key 62      ENDCALL             WAKE_DROPPED
key 229     MENU                WAKE_DROPPED

# 省略部分按键的对应内容
key 16      Q
key 17      W
key 18      E
key 19      R
key 20      T
key 115     VOLUME_UP
key 114     VOLUME_DOWN
```

在上述代码中，第 1 列为按键的扫描码，是一个整数值；第 2 列为按键的标签，是一个字符串，即完成了按键信息的第 1 次转换，将整型的扫描码转换成字符串类型的按键标签；第 3 列表示按键的 Flag，带有 WAKE 字符，表示此按键可以唤醒系统。

注意： 扫描码受驱动程序决定，不同的扫描码对应一个按键标签。两个手机的物理按键可以对应同一个功能按键。例如当上面的扫描码为 158 时，对应的标签为 BACK，经过第 2 次转换后，根据 KeycodeLabels.h 的 KEYCODES 数组可得出其对应的按键码是 4。

17.4.4　KCM 格式的按键字符映射文件

在 Android 系统中，KCM 格式文件是按键字符映射文件，用于表示按键字符的映射关系，功能是将整数类型按键码（keycode）转化成可以显示的字符。KCM 文件将被 makekcharmap 工具转换成二进制的格式，放在目标系统的/system/usr/keychars/目录下。

文件 qwerty.kcm 表示全键盘的字符映射关系，其主要代码如下所示。

```
[type=QWERTY]
# keycode       display   number   base   caps   fn    caps_fn
A               'A'       '2'      'a'    'A'    '#'   0x00
B               'B'       '2'      'b'    'B'    '<'   0x00
C               'C'       '2'      'c'    'C'    '9'   0x00E7
D               'D'       '3'      'd'    'D'    '5'   0x00
E               'E'       '3'      'e'    'E'    '2'   0x0301
F               'F'       '3'      'f'    'F'    '6'   0x00A5
G               'G'       '4'      'g'    'G'    '-'   '_'
H               'H'       '4'      'h'    'H'    '['   '{'
I               'I'       '4'      'i'    'I'    '$'   0x0302
J               'J'       '5'      'j'    'J'    ']'   '}'
K               'K'       '5'      'k'    'K'    '"'   '~'
L               'L'       '5'      'l'    'L'    '''   '`'
M               'M'       '6'      'm'    'M'    '!'   0x00
N               'N'       '6'      'n'    'N'    '>'   0x0303
```

在上述代码中，第一列表示转换之前的按键码，第二列及之后分别表示转换成的显示内容（display）和数字（number）等内容。这些转换的内容和文件 KeyCharacterMap.h 相对应，具体内容是在此文件的 getDisplayLabel()和 getNumber()等函数中定义的。

除了 QWERTY 映射类型之外，还可以映射 Q14（单键多字符对应的键盘）和 NUMERIC（12 键的数字键盘）。

17.4.5　分析文件 EventHub.cpp

在 Android 系统中，文件 frameworks/base/services/input/EventHub.cpp 是输入系统的核心控制文件，整个输入系统的主要功能都是在此文件中实现的。例如当按下电源键后，系统把 scanCode 写入对应的设备节点，文件 EventHub.cpp 会去读这个设备节点，并把 scanCode 通过 KL 文件对应成 keyCode 发送到上层。

在文件 EventHub.cpp 中需要定义设备节点所在的路径，定义代码如下所示。

```
static const char *WAKE_LOCK_ID = "KeyEvents";
static const char *DEVICE_PATH = "/dev/input";//输入设备的目录
```

在具体处理时，在函数 openPlatformInput()中通过调用函数 scan_dir()搜索路径下面所有 Input 驱动的设备节点。函数 scan_dir()会从目录中查找设备，找到后调用 open_device()函数以打开查找到的设备。其中函数 openPlatformInput()的实现代码如下所示。

```
bool EventHub::openPlatformInput(void)
{
    /*
     * Open platform-specific input device(s).
     */
    int res;

    mFDCount = 1;
    mFDs = (pollfd *)calloc(1, sizeof(mFDs[0]));
    mDevices = (device_t **)calloc(1, sizeof(mDevices[0]));
    mFDs[0].events = POLLIN;
    mDevices[0] = NULL;
#ifdef HAVE_INOTIFY
    mFDs[0].fd = inotify_init();
    res = inotify_add_watch(mFDs[0].fd, device_path, IN_DELETE | IN_CREATE);
    if(res < 0) {
        LOGE("could not add watch for %s, %s\n", device_path, strerror(errno));
    }
#else
    /*
     * The code in EventHub::getEvent assumes that mFDs[0] is an inotify fd.
     * We allocate space for it and set it to something invalid.
     */
    mFDs[0].fd = -1;
#endif

    res = scan_dir(device_path);
    if(res < 0) {
        LOGE("scan dir failed for %s\n", device_path);
        //open_device("/dev/input/event0");
    }

    return true;
}
```

再看函数 getEvent()，功能是在一个无限循环之内调用阻塞的函数等待事件到来，具体实现代码如下所示。



```
bool EventHub::getEvent(int32_t* outDeviceId, int32_t* outType,
        int32_t* outScancode, int32_t* outKeycode, uint32_t *outFlags,
        int32_t* outValue, nsecs_t* outWhen)
{
    *outDeviceId = 0;
    *outType = 0;
    *outScancode = 0;
    *outKeycode = 0;
    *outFlags = 0;
    *outValue = 0;
    *outWhen = 0;

    status_t err;

    fd_set readfds;
    int maxFd = -1;
    int cc;
    int i;
    int res;
    int pollres;
    struct input_event iev;

    // Note that we only allow one caller to getEvent(), so don't need
    // to do locking here...only when adding/removing devices.

    while(1) {

        // First, report any devices that had last been added/removed.
        if (mClosingDevices != NULL) {
            device_t* device = mClosingDevices;
            LOGV("Reporting device closed: id=0x%x, name=%s\n",
                 device->id, device->path.string());
            mClosingDevices = device->next;
            *outDeviceId = device->id;
            if (*outDeviceId == mFirstKeyboardId) *outDeviceId = 0;
            *outType = DEVICE_REMOVED;
            delete device;
            return true;
        }
        if (mOpeningDevices != NULL) {
            device_t* device = mOpeningDevices;
            LOGV("Reporting device opened: id=0x%x, name=%s\n",
                 device->id, device->path.string());
            mOpeningDevices = device->next;
            *outDeviceId = device->id;
            if (*outDeviceId == mFirstKeyboardId) *outDeviceId = 0;
            *outType = DEVICE_ADDED;
            return true;
        }

        release_wake_lock(WAKE_LOCK_ID);
```

```
pollres = poll(mFDs, mFDCount, -1);

acquire_wake_lock(PARTIAL_WAKE_LOCK, WAKE_LOCK_ID);

if (pollres <= 0) {
    if (errno != EINTR) {
        LOGW("select failed (errno=%d)\n", errno);
        usleep(100000);
    }
    continue;
}

//printf("poll %d, returned %d\n", mFDCount, pollres);
if(mFDs[0].revents & POLLIN) {
    read_notify(mFDs[0].fd);
}
for(i = 1; i < mFDCount; i++) {
    if(mFDs[i].revents) {
        LOGV("revents for %d = 0x%08x", i, mFDs[i].revents);
        if(mFDs[i].revents & POLLIN) {
            res = read(mFDs[i].fd, &iev, sizeof(iev));
            if (res == sizeof(iev)) {
                LOGV("%s got: t0=%d, t1=%d, type=%d, code=%d, v=%d",
                     mDevices[i]->path.string(),
                     (int) iev.time.tv_sec, (int) iev.time.tv_usec,
                     iev.type, iev.code, iev.value);
                *outDeviceId = mDevices[i]->id;
                if (*outDeviceId == mFirstKeyboardId) *outDeviceId = 0;
                *outType = iev.type;
                *outScancode = iev.code;
                if (iev.type == EV_KEY) {
                    err = mDevices[i]->layoutMap->map(iev.code, outKeycode, outFlags);
                    LOGV("iev.code=%d outKeycode=%d outFlags=0x%08x err=%d\n",
                        iev.code, *outKeycode, *outFlags, err);
                    if (err != 0) {
                        *outKeycode = 0;
                        *outFlags = 0;
                    }
                } else {
                    *outKeycode = iev.code;
                }
                *outValue = iev.value;
                *outWhen = s2ns(iev.time.tv_sec) + us2ns(iev.time.tv_usec);
                return true;
            } else {
                if (res<0) {
                    LOGW("could not get event (errno=%d)", errno);
                } else {
                    LOGE("could not get event (wrong size: %d)", res);
                }
```

```
                        continue;
                    }
                }
            }
        }
    }
}
```

在上述代码中，通过函数 poll()来阻塞程序的运行，此时为等待状态，不会开销内存。当 Input 设备的相应事件发生后会将函数 poll()返回，然后通过函数 read()读取 Input 设备发生的事件代码。

在 Android 系统中，有一些 Input 设备可能不需要经过 EventHub 处理，在这种情况下可以根据 EventHub 中的 open_device()函数进行处理。我们可以在驱动程序中设置一些标志来屏蔽一些设备。在函数 open_device()中实现了键盘、轨迹球和触摸屏等几种设备的处理功能，对其他设备可以忽略。此外还有另外一种简单的方法实现设备忽略功能，即将不需要 EventHub 处理的设备的设备节点放置在/dev/input 目录下。

另外，函数 open_device()还可以打开并处理 system/usr/keylayout/目录下的 KL 文件，具体实现代码如下所示。

```
const char* root = getenv("ANDROID_ROOT");
snprintf(keylayoutFilename, sizeof(keylayoutFilename),
         "%s/usr/keylayout/%s.kl", root, tmpfn);
bool defaultKeymap = false;
if (access(keylayoutFilename, R_OK)) {
    snprintf(keylayoutFilename, sizeof(keylayoutFilename),
        "%s/usr/keylayout/%s", root, "qwerty.kl");
    defaultKeymap = true;
}
```

注意： Android 中已经定义了丰富、完整的标准按键，一般情况下，我们只需要根据 KL 配置按键即可，不需要再为 Android 系统增加按键。当在现实项目中需要比较特殊的按键时，我们需要更改 Android 系统的框架层来实现更改按键功能。

在 Android 中增加新按键时，需要更改下面的文件。

- ☑ 文件 KeycodeLabels.h：保存在 frameworks/base/include/ui/目录下，需要修改 KeyCode 枚举数值和 KeycodeLabel 类型 Code 数组。
- ☑ 文件 KeyEvent.Java：保存在 frameworks/base/core/Java/android/view/目录下，在此可以定义整数值作为平台的 API 供 Java 应用程序使用。
- ☑ 文件 attrs.xml：保存在 frameworks/base/core/res/res/values/目录下，表示属性的资源文件，需要修改其中的 name="keycode"的 attr。框架层增加完成后，只需要更改 KL 文件，增加按键的映射关系即可。

除此之外，还有一种更为简易的做法，就是使用 Android 中已经定义的“特殊”按键码作为这个新增按键的键码。使用这种方式 Android 的框架层不需要做任何改动。这种方式的潜在问题是当某些第三方的应用可能已经使用那些特殊按键时，会意外激发系统的这种新增的按键。

17.5　实战演练

通过本章前面内容的讲解，已经了解了 Android 输入系统的内核和硬件抽象层的具体实现过程。本节将

依次讲解 Android 内置的模拟器、MSM 内核和 OMAP 内核中实现输入驱动的具体流程。

17.5.1　在内置模拟器中实现输入驱动

在 Goldfish 虚拟处理器中，使用 event 驱动程序作为键盘输入功能的驱动程序，其驱动程序的相关文件是 drivers/input/keyboard/goldfish_events.c。此驱动程序是一个标准的 event 驱动程序，在用户空间的设备节点为/dev/event/event0，其核心代码如下所示。

```
static irqreturn_t events_interrupt(int irq, void *dev_id)
{
	struct event_dev *edev = dev_id;
	unsigned type, code, value;
	type = __raw_readl(edev->addr + REG_READ);		//类型
	code = __raw_readl(edev->addr + REG_READ);		//码
	value = __raw_readl(edev->addr + REG_READ);		//数值
	input_event(edev->input, type, code, value);
	return IRQ_HANDLED;
}
```

函数 events_interrupt()是按键事件的中断处理函数，当中断发生后会读取虚拟寄存器的内容，并将信息上报。模拟器根据主机环境键盘按下的情况可以得到虚拟寄存器中的内容。

在模拟器环境中，使用默认的所有的 KL 和 KCM 文件，由于模拟器环境支持全键盘，因此基本上包含了大部分的功能。在模拟器环境中，实际上按键的扫描码对应的是桌面电脑的键盘（效果和鼠标点击模拟器的控制面板类似）。当按下键盘的某些按键后会转换为驱动程序中的扫描码，然后再由上层的用户空间处理。上述过程和实际系统中是类似的。通过更改默认 KL 文件的方式，又可以更改实际按键的映射关系。

17.5.2　在 MSM 高通处理器中实现输入驱动

在高通 MSM 的 mahimahi 平台中，具有触摸屏、轨迹球和简单按键功能，这些功能是由 Android 系统中驱动程序实现的，并且需要用户空间的内容来协助实现。

在 mahimahi 平台中，输入系统设备包括以下 Event 设备。

☑　/dev/input/event4：几个按键的设备。
☑　/dev/input/event2：触摸屏设备。
☑　/dev/input/event5：轨迹球设备。

1．触摸屏驱动

高通 mahimahi 平台的触摸屏驱动程序的实现文件是 drivers/input/touchscreen/synaptics_i2c_rmi.c，此文件的核心是函数 synaptics_ts_probe()，在该函数中需要进行触摸屏工作模式的初始化，对作为输入设备的触摸屏驱动在 Linux 平台下的设备名注册，同时初始化触摸事件触发时引起的中断操作。此函数的实现代码如下所示。

```
static int synaptics_ts_probe(
	struct i2c_client *client, const struct i2c_device_id *id)
{
	struct synaptics_ts_data *ts;
	uint8_t buf0[4];
	uint8_t buf1[8];
	struct i2c_msg msg[2];
```

```
int ret = 0;
uint16_t max_x, max_y;
int fuzz_x, fuzz_y, fuzz_p, fuzz_w;
struct synaptics_i2c_rmi_platform_data *pdata;
int inactive_area_left;
int inactive_area_right;
int inactive_area_top;
int inactive_area_bottom;
int snap_left_on;
int snap_left_off;
int snap_right_on;
int snap_right_off;
int snap_top_on;
int snap_top_off;
int snap_bottom_on;
int snap_bottom_off;
uint32_t panel_version;

if (!i2c_check_functionality(client->adapter, I2C_FUNC_I2C)) {
        printk(KERN_ERR "synaptics_ts_probe: need I2C_FUNC_I2C\n");
        ret = -ENODEV;
        goto err_check_functionality_failed;
}

ts = kzalloc(sizeof(*ts), GFP_KERNEL);
if (ts == NULL) {
        ret = -ENOMEM;
        goto err_alloc_data_failed;
}
INIT_WORK(&ts->work, synaptics_ts_work_func);
ts->client = client;
i2c_set_clientdata(client, ts);
pdata = client->dev.platform_data;
if (pdata)
        ts->power = pdata->power;
if (ts->power) {
        ret = ts->power(1);
        if (ret < 0) {
                printk(KERN_ERR "synaptics_ts_probe power on failed\n");
                goto err_power_failed;
        }
}

ret = i2c_smbus_write_byte_data(ts->client, 0xf4, 0x01); /* device command = reset */
if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_write_byte_data failed\n");
        /* fail? */
}
{
        int retry = 10;
        while (retry-- > 0) {
                ret = i2c_smbus_read_byte_data(ts->client, 0xe4);
```

```
                if (ret >= 0)
                        break;
                msleep(100);
        }
}
if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_byte_data failed\n");
        goto err_detect_failed;
}
printk(KERN_INFO "synaptics_ts_probe: Product Major Version %x\n", ret);
panel_version = ret << 8;
ret = i2c_smbus_read_byte_data(ts->client, 0xe5);
if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_byte_data failed\n");
        goto err_detect_failed;
}
printk(KERN_INFO "synaptics_ts_probe: Product Minor Version %x\n", ret);
panel_version |= ret;

ret = i2c_smbus_read_byte_data(ts->client, 0xe3);
if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_byte_data failed\n");
        goto err_detect_failed;
}
printk(KERN_INFO "synaptics_ts_probe: product property %x\n", ret);

if (pdata) {
        while (pdata->version > panel_version)
                pdata++;
        ts->flags = pdata->flags;
        inactive_area_left = pdata->inactive_left;
        inactive_area_right = pdata->inactive_right;
        inactive_area_top = pdata->inactive_top;
        inactive_area_bottom = pdata->inactive_bottom;
        snap_left_on = pdata->snap_left_on;
        snap_left_off = pdata->snap_left_off;
        snap_right_on = pdata->snap_right_on;
        snap_right_off = pdata->snap_right_off;
        snap_top_on = pdata->snap_top_on;
        snap_top_off = pdata->snap_top_off;
        snap_bottom_on = pdata->snap_bottom_on;
        snap_bottom_off = pdata->snap_bottom_off;
        fuzz_x = pdata->fuzz_x;
        fuzz_y = pdata->fuzz_y;
        fuzz_p = pdata->fuzz_p;
        fuzz_w = pdata->fuzz_w;
} else {
        inactive_area_left = 0;
        inactive_area_right = 0;
        inactive_area_top = 0;
        inactive_area_bottom = 0;
        snap_left_on = 0;
```

```
        snap_left_off = 0;
        snap_right_on = 0;
        snap_right_off = 0;
        snap_top_on = 0;
        snap_top_off = 0;
        snap_bottom_on = 0;
        snap_bottom_off = 0;
        fuzz_x = 0;
        fuzz_y = 0;
        fuzz_p = 0;
        fuzz_w = 0;
}

ret = i2c_smbus_read_byte_data(ts->client, 0xf0);
if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_byte_data failed\n");
        goto err_detect_failed;
}
printk(KERN_INFO "synaptics_ts_probe: device control %x\n", ret);

ret = i2c_smbus_read_byte_data(ts->client, 0xf1);
if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_byte_data failed\n");
        goto err_detect_failed;
}
printk(KERN_INFO "synaptics_ts_probe: interrupt enable %x\n", ret);

ret = i2c_smbus_write_byte_data(ts->client, 0xf1, 0); /* disable interrupt */
if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_write_byte_data failed\n");
        goto err_detect_failed;
}

msg[0].addr = ts->client->addr;
msg[0].flags = 0;
msg[0].len = 1;
msg[0].buf = buf0;
buf0[0] = 0xe0;
msg[1].addr = ts->client->addr;
msg[1].flags = I2C_M_RD;
msg[1].len = 8;
msg[1].buf = buf1;
ret = i2c_transfer(ts->client->adapter, msg, 2);
if (ret < 0) {
        printk(KERN_ERR "i2c_transfer failed\n");
        goto err_detect_failed;
}
printk(KERN_INFO "synaptics_ts_probe: 0xe0: %x %x %x %x %x %x %x %x\n",
            buf1[0], buf1[1], buf1[2], buf1[3],
            buf1[4], buf1[5], buf1[6], buf1[7]);

ret = i2c_smbus_write_byte_data(ts->client, 0xff, 0x10); /* page select = 0x10 */
```

```
if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_write_byte_data failed for page select\n");
        goto err_detect_failed;
}
ret = i2c_smbus_read_word_data(ts->client, 0x04);
if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_word_data failed\n");
        goto err_detect_failed;
}
ts->max[0] = max_x = (ret >> 8 & 0xff) | ((ret & 0x1f) << 8);
ret = i2c_smbus_read_word_data(ts->client, 0x06);
if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_word_data failed\n");
        goto err_detect_failed;
}
ts->max[1] = max_y = (ret >> 8 & 0xff) | ((ret & 0x1f) << 8);
if (ts->flags & SYNAPTICS_SWAP_XY)
        swap(max_x, max_y);

ret = synaptics_init_panel(ts); /* will also switch back to page 0x04 */
if (ret < 0) {
        printk(KERN_ERR "synaptics_init_panel failed\n");
        goto err_detect_failed;
}

ts->input_dev = input_allocate_device();//创建设备
if (ts->input_dev == NULL) {
        ret = -ENOMEM;
        printk(KERN_ERR "synaptics_ts_probe: Failed to allocate input device\n");
        goto err_input_dev_alloc_failed;
}
ts->input_dev->name = "synaptics-rmi-touchscreen";
//声明输入设备
set_bit(EV_SYN, ts->input_dev->evbit);
set_bit(EV_KEY, ts->input_dev->evbit);
set_bit(BTN_TOUCH, ts->input_dev->keybit);
set_bit(BTN_2, ts->input_dev->keybit);
set_bit(EV_ABS, ts->input_dev->evbit);
inactive_area_left = inactive_area_left * max_x / 0x10000;
inactive_area_right = inactive_area_right * max_x / 0x10000;
inactive_area_top = inactive_area_top * max_y / 0x10000;
inactive_area_bottom = inactive_area_bottom * max_y / 0x10000;
snap_left_on = snap_left_on * max_x / 0x10000;
snap_left_off = snap_left_off * max_x / 0x10000;
snap_right_on = snap_right_on * max_x / 0x10000;
snap_right_off = snap_right_off * max_x / 0x10000;
snap_top_on = snap_top_on * max_y / 0x10000;
snap_top_off = snap_top_off * max_y / 0x10000;
snap_bottom_on = snap_bottom_on * max_y / 0x10000;
snap_bottom_off = snap_bottom_off * max_y / 0x10000;
fuzz_x = fuzz_x * max_x / 0x10000;
fuzz_y = fuzz_y * max_y / 0x10000;
```

```
	ts->snap_down[!!(ts->flags & SYNAPTICS_SWAP_XY)] = -inactive_area_left;
	ts->snap_up[!!(ts->flags & SYNAPTICS_SWAP_XY)] = max_x + inactive_area_right;
	ts->snap_down[!(ts->flags & SYNAPTICS_SWAP_XY)] = -inactive_area_top;
	ts->snap_up[!(ts->flags & SYNAPTICS_SWAP_XY)] = max_y + inactive_area_bottom;
	ts->snap_down_on[!!(ts->flags & SYNAPTICS_SWAP_XY)] = snap_left_on;
	ts->snap_down_off[!!(ts->flags & SYNAPTICS_SWAP_XY)] = snap_left_off;
	ts->snap_up_on[!!(ts->flags & SYNAPTICS_SWAP_XY)] = max_x - snap_right_on;
	ts->snap_up_off[!!(ts->flags & SYNAPTICS_SWAP_XY)] = max_x - snap_right_off;
	ts->snap_down_on[!(ts->flags & SYNAPTICS_SWAP_XY)] = snap_top_on;
	ts->snap_down_off[!(ts->flags & SYNAPTICS_SWAP_XY)] = snap_top_off;
	ts->snap_up_on[!(ts->flags & SYNAPTICS_SWAP_XY)] = max_y - snap_bottom_on;
	ts->snap_up_off[!(ts->flags & SYNAPTICS_SWAP_XY)] = max_y - snap_bottom_off;
	printk(KERN_INFO "synaptics_ts_probe: max_x %d, max_y %d\n", max_x, max_y);
	printk(KERN_INFO "synaptics_ts_probe: inactive_x %d %d, inactive_y %d %d\n",
	       inactive_area_left, inactive_area_right,
	       inactive_area_top, inactive_area_bottom);
	printk(KERN_INFO "synaptics_ts_probe: snap_x %d-%d %d-%d, snap_y %d-%d %d-%d\n",
	       snap_left_on, snap_left_off, snap_right_on, snap_right_off,
	       snap_top_on, snap_top_off, snap_bottom_on, snap_bottom_off);
	//配置具体事件
	input_set_abs_params(ts->input_dev, ABS_X, -inactive_area_left, max_x + inactive_area_right, fuzz_x, 0);
	input_set_abs_params(ts->input_dev, ABS_Y, -inactive_area_top, max_y + inactive_area_bottom, fuzz_y, 0);
	input_set_abs_params(ts->input_dev, ABS_PRESSURE, 0, 255, fuzz_p, 0);
	input_set_abs_params(ts->input_dev, ABS_TOOL_WIDTH, 0, 15, fuzz_w, 0);
	input_set_abs_params(ts->input_dev,  ABS_HAT0X,  -inactive_area_left,  max_x  +  inactive_area_right,
fuzz_x, 0);
	input_set_abs_params(ts->input_dev,  ABS_HAT0Y,  -inactive_area_top,  max_y  +  inactive_area_bottom,
fuzz_y, 0);
	/* ts->input_dev->name = ts->keypad_info->name; */
	ret = input_register_device(ts->input_dev);
	if (ret) {
		printk(KERN_ERR "synaptics_ts_probe: Unable to register %s input device\n", ts->input_dev->name);
		goto err_input_register_device_failed;
	}
	if (client->irq) {
		ret = request_irq(client->irq, synaptics_ts_irq_handler, 0, client->name, ts);
		if (ret == 0) {
			ret = i2c_smbus_write_byte_data(ts->client, 0xf1, 0x01); /* enable abs int */
			if (ret)
				free_irq(client->irq, ts);
		}
		if (ret == 0)
			ts->use_irq = 1;
		else
			dev_err(&client->dev, "request_irq failed\n");
	}
	if (!ts->use_irq) {
		hrtimer_init(&ts->timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
		ts->timer.function = synaptics_ts_timer_func;
		hrtimer_start(&ts->timer, ktime_set(1, 0), HRTIMER_MODE_REL);
	}
#ifdef CONFIG_HAS_EARLYSUSPEND
```

```
	ts->early_suspend.level = EARLY_SUSPEND_LEVEL_BLANK_SCREEN + 1;
	ts->early_suspend.suspend = synaptics_ts_early_suspend;
	ts->early_suspend.resume = synaptics_ts_late_resume;
	register_early_suspend(&ts->early_suspend);
#endif

	printk(KERN_INFO "synaptics_ts_probe: Start touchscreen %s in %s mode\n", ts->input_dev->name,
ts->use_irq ? "interrupt" : "polling");

	return 0;

err_input_register_device_failed:
	input_free_device(ts->input_dev);

err_input_dev_alloc_failed:
err_detect_failed:
err_power_failed:
	kfree(ts);
err_alloc_data_failed:
err_check_functionality_failed:
	return ret;
}
```

在上述代码中，通过 i2c_smbus_read_byte_data()函数对其寄存器信息进行读取即可完成其事件信息的获取，也可以通过 i2c_transfer 完成对其寄存器信息的批量读取。

2．按键和轨迹球驱动

MSM 具有按键和轨迹球的功能，对应的驱动程序在文件 arch/arm/mach-msm/board-mahimahi-keypad.c 中，接下来开始介绍此文件的实现流程。

（1）文件 board-mahimahi-keypad.c 中的全局定义代码如下所示。

```
static struct gpio_event_info *mahimahi_input_info[] = {
	&mahimahi_keypad_matrix_info.info,		//键盘矩阵
	&mahimahi_keypad_key_info.info,			//键盘信息
	&jogball_x_axis.info.info,			//轨迹球 X 方向信息
	&jogball_y_axis.info.info,			//轨迹球 Y 方向信息
};
static struct gpio_event_platform_data mahimahi_input_data = {
	.names = {
		"mahimahi-keypad",			//按键设备
		"mahimahi-nav",				//轨迹球设备
		NULL,
	},
	.info = mahimahi_input_info,
	.info_count = ARRAY_SIZE(mahimahi_input_info),
	.power = jogball_power,
};
static struct platform_device mahimahi_input_device = {
	.name = GPIO_EVENT_DEV_NAME,
	.id = 0,
	.dev = {
```

```
            .platform_data = &mahimahi_input_data,
        },
};
```

因为按键和轨迹球是通过 GPIO 系统来实现的，所以在上面定义了一个 gpio_event_info 类型的数组。mahimahi-keypad 和 mahimahi-nav 分别表示两个设备的名称。在 gpio_event_info 指针数组 mahimahi_input_info 中包括 4 个成员，分别是 mahimahi_keypad_matrix_info.info、mahimahi_keypad_key_info.info、jogball_x_axis.info.info 和 jogball_y_axis.info.info。

（2）使用 gpio_event_matrix_info 矩阵定义按键驱动，此驱动是利用 GPIO 矩阵实现的，在定义时需要包含按键的 GPIO 矩阵和 Input 设备的信息，具体代码如下所示。

```
static unsigned int mahimahi_col_gpios[] = { 33, 32, 31 };
static unsigned int mahimahi_row_gpios[] = { 42, 41, 40 };

#define KEYMAP_INDEX(col, row)   ((col)*ARRAY_
SIZE(mahimahi_row_gpios) + (row))
#define KEYMAP_SIZE (ARRAY_SIZE(mahimahi_col_gpios) * \
                     ARRAY_SIZE(mahimahi_row_gpios))
static const unsigned short mahimahi_keymap
[KEYMAP_SIZE] = {               //按键映射关系
    [KEYMAP_INDEX(0, 0)] = KEY_VOLUMEUP, /* 115 */
    [KEYMAP_INDEX(0, 1)] = KEY_VOLUMEDOWN, /* 114 */
    [KEYMAP_INDEX(1, 1)] = MATRIX_KEY(1, BTN_MOUSE),
};
static struct gpio_event_matrix_info mahimahi
_keypad_matrix_info = {
    .info.func = gpio_event_matrix_func,
//关键函数实现
    .keymap = mahimahi_keymap,
    .output_gpios = mahimahi_col_gpios,
    .input_gpios = mahimahi_row_gpios,
    .noutputs = ARRAY_SIZE(mahimahi_col_gpios),
    .ninputs = ARRAY_SIZE(mahimahi_row_gpios),
    .settle_time.tv.nsec = 40 * NSEC_PER_USEC,
    .poll_time.tv.nsec = 20 * NSEC_PER_MSEC,
    .flags = (GPIOKPF_LEVEL_TRIGGERED_IRQ |
            GPIOKPF_REMOVE_PHANTOM_KEYS |
            GPIOKPF_PRINT_UNMAPPED_KEYS),
};
static struct gpio_event_direct_entry mahimahi_
keypad_key_map[] = { //Power 按键
    {
        .gpio = MAHIMAHI_GPIO_POWER_KEY,
        .code = KEY_POWER,
    },
};
static struct gpio_event_input_info mahimahi_
keypad_key_info = {
    .info.func = gpio_event_input_func,
//关键函数实现
```

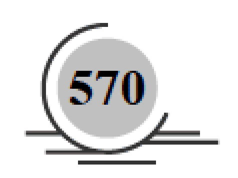
```
        .info.no_suspend = true,
        .flags = 0,
        .type = EV_KEY,
        .keymap = mahimahi_keypad_key_map,
        .keymap_size = ARRAY_SIZE(mahimahi_keypad_key_map)
};
```

在上述代码中，mahimahi_keypad_key_matrix_info 和 mahimahi_keypad_info 是 gpio_event_matrix_info 类型的结构体，分别实现两个按键和一个按键的处理功能。

（3）使用 GPIO 驱动实现轨迹球部分驱动，在实现时由 X 方向和 Y 方向两部分组成。具体代码如下所示。

```
static uint32_t jogball_x_gpios[] = {
    MAHIMAHI_GPIO_BALL_LEFT, MAHIMAHI_GPIO_BALL_RIGHT,
};
static uint32_t jogball_y_gpios[] = {
    MAHIMAHI_GPIO_BALL_UP, MAHIMAHI_GPIO_BALL_DOWN,
};
static struct jog_axis_info jogball_x_axis = {
//X 轴的内容
    .info = {
        .info.func = gpio_event_axis_func,
//关键函数实现
        .count = ARRAY_SIZE(jogball_x_gpios),
        .dev = 1,
        .type = EV_REL,
        .code = REL_X,
        .decoded_size = 1U << ARRAY_SIZE(jogball_x_gpios),
        .map = jogball_axis_map,
        .gpio = jogball_x_gpios,
        .flags = GPIOEAF_PRINT_UNKNOWN_DIRECTION,
    }
};
static struct jog_axis_info jogball_y_axis = {
//Y 轴的内容
    .info = {
        .info.func = gpio_event_axis_func,
//关键函数实现
        .count = ARRAY_SIZE(jogball_y_gpios)
        .dev = 1,
        .type = EV_REL,
        .code = REL_Y,
        .decoded_size = 1U << ARRAY_SIZE(jogball_y_gpios),
        .map = jogball_axis_map,
        .gpio = jogball_y_gpios,
        .flags = GPIOEAF_PRINT_UNKNOWN_DIRECTION,
    }
};
```

在上述代码中，使用 jog_axis_info 类型的结构体定义了轨迹球，这种设备的类型（type）是相对设备 EV_REL。



注意： 除了默认的 AVRCP.kl 和 qwerty.kl 之外，在高通 mahimahi 平台中新增了文件 h2w_headset.kl 和 mahimahi-keypad.kl。

17.5.3　在 Zoom 平台中实现输入驱动

1．触摸屏驱动程序

OMAP 的 Zoom 平台的输入设备包括触摸屏和键盘（Qwerty 全键盘）两种，其中触摸屏驱动程序保存在文件 drivers/input/touchscreen/synaptics_i2c_rmi.c 中，这是一个 I2C 的触摸屏的驱动程序，和 MSM 完全相同，此处将不再进行讲解。

2．键盘驱动程序

在 Zoom 平台中，键盘驱动程序保存在文件 drivers/input/keyboard/twl4030_keypad.c 中，此驱动使用了 I2C 的接口，驱动本身经过了一次封装处理。文件 twl4030_keypad.c 中的核心内容是中断处理相关的内容，其中函数 do_kp_irq()是标准 Linux 系统的中断处理函数，其主要代码如下所示。

```
static irqreturn_t do_kp_irq(int irq, void *_kp)
{
    struct twl4030_keypad *kp = _kp;
    u8 reg;
    int ret;
    ret = twl4030_kpread(kp, &reg, KEYP_ISR1, 1);
//调用 twl4030_i2c_read
    if ((ret >= 0) && (reg & KEYP_IMR1_KP))
        twl4030_kp_scan(kp, 0);
//非释放所有的处理
    else
        twl4030_kp_scan(kp, 1);
//释放所有的处理
    return IRQ_HANDLED;
}
```

函数 twl4030_kp_scan()负责实现核心处理功能，先负责找到按键的行列，然后调用函数 input_ report_key()来汇报结果。函数 twl4030_kp_scan()的主要实现代码如下所示。

```
static void twl4030_kp_scan(struct twl4030_keypad *kp, int release_all)
{
    u16 new_state[MAX_ROWS];
    int col, row;
    //...省略部分内容
    for (row = 0; row < kp->n_rows; row++) {
        int changed = new_state[row] ^ kp->kp_state[row];
    //...省略部分内容
        for (col = 0; col < kp->n_cols; col++) {
            int key;
            key = twl4030_find_key(kp, col, row);
    //...省略部分内容
            input_report_key(kp->input, key,            //上报按键消息
                    new_state[row] & (1 << col));
        }
        kp->kp_state[row] = new_state[row];
```

```
	}
	input_sync(kp->input);
}
```

接下来使用函数 twl4030_find_key()根据行列来扫描键盘信息，实现代码如下所示。

```
static int twl4030_find_key(struct
twl4030_keypad *kp, int col, int row)
{
	int i, rc;
	rc = KEY(col, row, 0);
	for (i = 0; i < kp->keymapsize; i++)
		if ((kp->keymap[i] & ROWCOL_MASK) == rc)
			return kp->keymap[i] &
(KEYNUM_MASK | KEY_PERSISTENT);
	return -EINVAL;
}
```

需要注意，上述代码中使用 kp->keymap 数组定义了按键映射关系，此数组在文件 arch/arm/mach-omap2/board-zoom2.c 中定义，并对应于数组 zoom2_twl4030_keymap，此数组的定义代码如下所示。

```
static int zoom2_twl4030_keymap[] = {
	KEY(0, 0, KEY_E),
	KEY(1, 0, KEY_R),
	KEY(2, 0, KEY_T),
	KEY(3, 0, KEY_HOME),
	KEY(6, 0, KEY_I),
	KEY(7, 0, KEY_LEFTSHIFT),
	...
	KEY(7, 7, KEY_DOWN),
	KEY(0, 7, KEY_PROG1),
	KEY(1, 7, KEY_PROG2),
	KEY(2, 7, KEY_PROG3),
	KEY(3, 7, KEY_PROG4),
	0
};
```

在 OMAP 的 Zoom 平台中，因为键盘基本上是全键盘，并且其数字键和字母键是共用的，所以使用全键盘的配置文件基本上可以实现全部功能。

第 18 章　LCD 显示驱动

在现实开发应用中，LCD 显示系统驱动的功能是操作显示设备并获得显示终端。在 Android 平台中，LCD 显示驱动是通过 FrameBuffer 技术实现的，所以通常将 LCD 驱动称为 FrameBuffer 驱动。本章将详细讲解 LCD 显示系统驱动的基本架构知识，为读者学习本书后面的知识打下基础。

18.1　LCD 系统介绍

在 Linux 系统中，标准的设备显示驱动是 FrameBuffer。直观地说，FrameBuffer 驱动程序是 PC 系统中的显卡驱动程序，是嵌入式系统中的 SOC 处理器，通常将 FrameBuffer 作为其 LCD 控制器或者其他显示设备的驱动。在 Android 系统中，显示系统对应了硬件层中的 LCD、LCD 控制器和 VGA 输出设备等显示设备。另外，显示系统和 Surface 库有着很大的联系，在显示系统的下层实现了对基本显示输出的封装，并且在 Surface 中的部分库还提供了对多个图层的支持。

Framebuffer 驱动是一个字符设备，此驱动通常在文件系统的设备节点中，位置为/dev/fbX。

FrameBuffer 驱动的主设备号是 29，次设备号由递增数字生成。每个系统可以有多个显示设备，分别使用/dev/fb0、/dev/fb1 和/dev/fb2 等来表示。

在用户空间中，通常 FrameBuffer 驱动使用 ioctl 和 mmap 等文件系统接口进行操作，其中 ioctl 用于获得和设置信息，mmap 可以将 FrameBuffer 的内存映射到用户空间。另外 FrameBuffer 驱动直接支持 Write 操作，可以直接用写的方式输出显示内容。显示驱动 FrameBuffer 的架构图如图 18-1 所示。

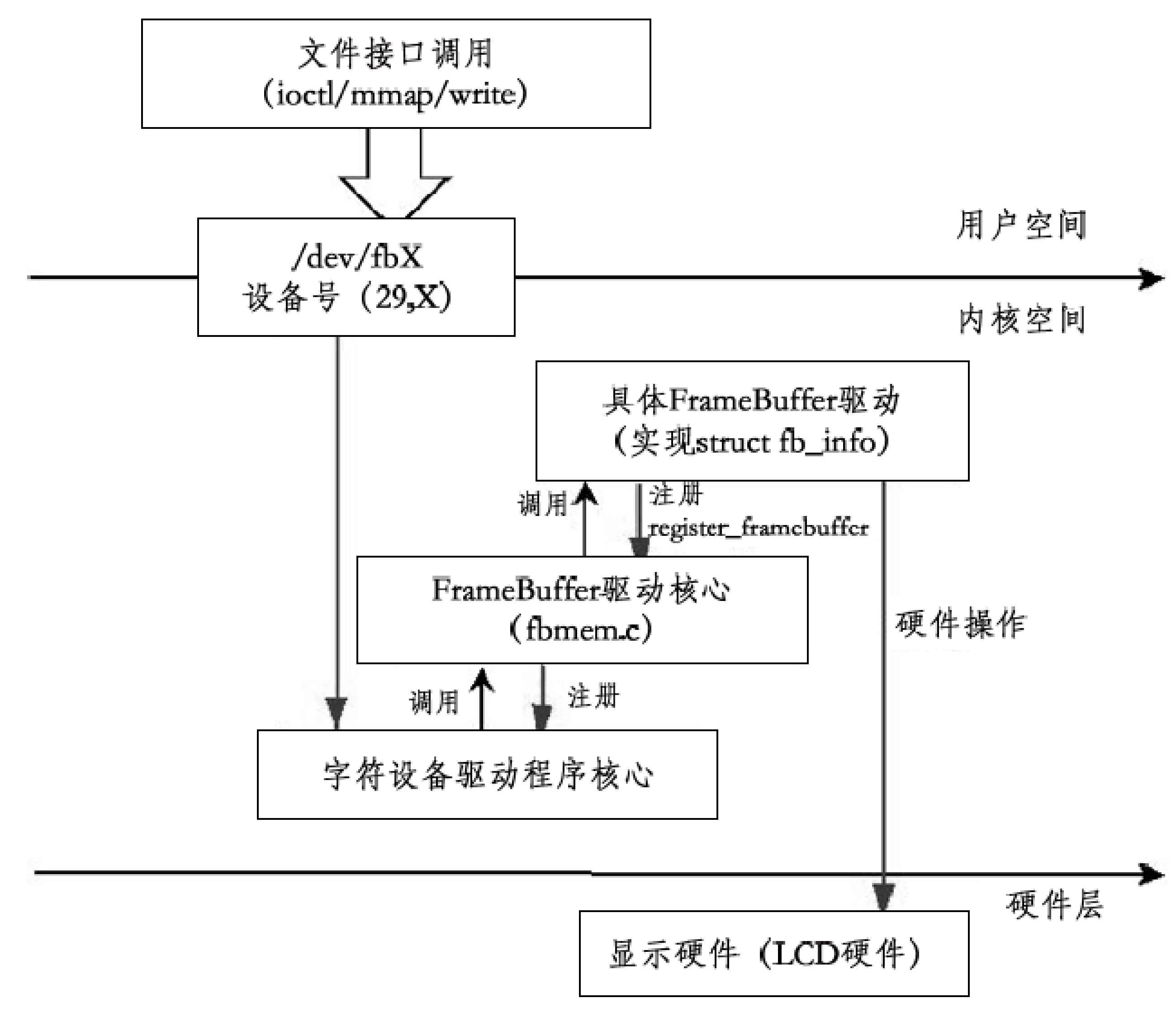


图 18-1　显示驱动 FrameBuffer 的架构图

18.2　FrameBuffer 内核层详解

在 Linux 内核系统中，FrameBuffer 驱动主要涉及如下两个文件。

☑　include/linux/fb.h：是 FrameBuffer 驱动的头文件。

☑ drivers/video/fbmem.c：是 FrameBuffer 驱动的核心实现文件。

下面将详细讲解上述两个文件的具体实现流程。

18.2.1 分析接口文件 fb.h

在文件 fb.h 中，首先定义了 FrameBuffer 驱动中核心的数据接口是 fb_info，具体实现代码如下所示。

```
struct fb_info {
        atomic_t count;
        int node;
        int flags;
        struct mutex lock;                  /* Lock for open/release/ioctl funcs */
        struct mutex mm_lock;               /* Lock for fb_mmap and smem_* fields */
        struct fb_var_screeninfo var;       /*  显示屏的信息 */
        struct fb_fix_screeninfo fix;       /*  显示屏的固定信息 */
        struct fb_monspecs monspecs;        /* Current Monitor specs */
        struct work_struct queue;           /* Framebuffer event queue */
        struct fb_pixmap pixmap;            /* Image hardware mapper */
        struct fb_pixmap sprite;            /* Cursor hardware mapper */
        struct fb_cmap cmap;                /* Current cmap */
        struct list_head modelist;          /* mode list */
        struct fb_videomode *mode;          /* current mode */
```

在上述数据接口 fb_info 中，包含了 FrameBuffer 驱动的主要信息。具体说明如下所示。

☑ struct fb_var_screeninfo 和 struct fb_fix_screeninfo：是两个相关的数据结构，分别对应 FBIOGET_VSCREENINFO 和 FBIOGET_FSCREENINFO 两个 ioctl，用于从用户空间获得显示信息。

其中结构 fb_var_screeninfo 用于记录用户可修改的显示控制器参数，包括屏幕的分辨率和每个像素点的比特数。结构 fb_var_screeninfo 的具体实现代码如下所示。

```
struct fb_var_screeninfo {
__u32 xres; /* visible resolution */
__u32 yres;
__u32 xres_virtual; /* virtual resolution */
__u32 yres_virtual;
__u32 xoffset; /* offset from virtual to visible */
__u32 yoffset; /* resolution */

__u32 bits_per_pixel; /* guess what */
__u32 grayscale; /* != 0 Graylevels instead of colors */

struct fb_bitfield red; /* bitfield in fb mem if true color, */
struct fb_bitfield green; /* else only length is significant */
struct fb_bitfield blue;
struct fb_bitfield transp; /* transparency */

__u32 nonstd; /* != 0 Non standard pixel format */

__u32 activate; /* see FB_ACTIVATE_* */

__u32 height; /* height of picture in mm       */
__u32 width; /* width of picture in mm        */
```

```
__u32 accel_flags; /* (OBSOLETE) see fb_info.flags */

/* Timing: All values in pixclocks, except pixclock (of course) */
__u32 pixclock; /* pixel clock in ps (pico seconds) */
__u32 left_margin; /* time from sync to picture */
__u32 right_margin; /* time from picture to sync */
__u32 upper_margin; /* time from sync to picture */
__u32 lower_margin;
__u32 hsync_len; /* length of horizontal sync */
__u32 vsync_len; /* length of vertical sync */
__u32 sync; /* see FB_SYNC_* */
__u32 vmode; /* see FB_VMODE_* */
__u32 rotate; /* angle we rotate counter clockwise */
__u32 reserved[5]; /* Reserved for future compatibility */
}
```

而结构 fb_fix_screeninfo 记录了用户不能修改的显示控制器参数，例如显示缓存的物理地址。结构 fb_fix_screeninfo 的具体实现代码如下所示。

```
struct fb_fix_screeninfo {
char id[16]; /* identification string eg "TT Builtin" */
unsigned long smem_start; /* Start of frame buffer mem */
/* (physical address) */
__u32 smem_len; /* Length of frame buffer mem */
__u32 type; /* see FB_TYPE_* */
__u32 type_aux; /* Interleave for interleaved Planes */
__u32 visual; /* see FB_VISUAL_* */
__u16 xpanstep; /* zero if no hardware panning   */
__u16 ypanstep; /* zero if no hardware panning   */
__u16 ywrapstep; /* zero if no hardware ywrap    */
__u32 line_length; /* length of a line in bytes     */
unsigned long mmio_start; /* Start of Memory Mapped I/O    */
/* (physical address) */
__u32 mmio_len; /* Length of Memory Mapped I/O   */
__u32 accel; /* Indicate to driver which */
/*   specific chip/card we have */
__u16 reserved[3]; /* Reserved for future compatibility */
};
```

☑ 结构 fb_ops：表示 FrameBuffer 驱动的操作，是一个类似于 file_operations 的可实现文件设备操作的数据结构。结构 fb_ops 的具体实现代码如下所示。

```
struct fb_ops {
/* open/release and usage marking */
struct module *owner;
int (*fb_open)(struct fb_info *info, int user);
int (*fb_release)(struct fb_info *info, int user);

/* For framebuffers with strange non linear layouts or that do not
* work with normal memory mapped access
*/
ssize_t (*fb_read)(struct fb_info *info, char __user *buf,
  size_t count, loff_t *ppos);
ssize_t (*fb_write)(struct fb_info *info, const char __user *buf,
```

```
   size_t count, loff_t *ppos);

/* checks var and eventually tweaks it to something supported,
* DO NOT MODIFY PAR */
int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info *info);

/* set the video mode according to info->var */
int (*fb_set_par)(struct fb_info *info);

/* set color register */
int (*fb_setcolreg)(unsigned regno, unsigned red, unsigned green,
     unsigned blue, unsigned transp, struct fb_info *info);

/* set color registers in batch */
int (*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);

/* blank display */
int (*fb_blank)(int blank, struct fb_info *info);

/* pan display */
int (*fb_pan_display)(struct fb_var_screeninfo *var, struct fb_info *info);

/* Draws a rectangle */
void (*fb_fillrect) (struct fb_info *info, const struct fb_fillrect *rect);
/* Copy data from area to another */
void (*fb_copyarea) (struct fb_info *info, const struct fb_copyarea *region);
/* Draws a image to the display */
void (*fb_imageblit) (struct fb_info *info, const struct fb_image *image);

/* Draws cursor */
int (*fb_cursor) (struct fb_info *info, struct fb_cursor *cursor);

/* Rotates the display */
void (*fb_rotate)(struct fb_info *info, int angle);

/* wait for blit idle, optional */
int (*fb_sync)(struct fb_info *info);

/* perform fb specific ioctl (optional) */
int (*fb_ioctl)(struct fb_info *info, unsigned int cmd,
unsigned long arg);

/* Handle 32bit compat ioctl (optional) */
int (*fb_compat_ioctl)(struct fb_info *info, unsigned cmd,
unsigned long arg);

/* perform fb specific mmap */
int (*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma);

/* save current hardware state */
void (*fb_save_state)(struct fb_info *info);
```

```
/* restore saved state */
void (*fb_restore_state)(struct fb_info *info);

/* get capability given var */
void (*fb_get_caps)(struct fb_info *info, struct fb_blit_caps *caps,
    struct fb_var_screeninfo *var);
};
```

18.2.2　内核实现文件

（1）在具体实现 FrameBuffer 驱动的过程中，通常使用如下两个函数分别实现注册和注销功能。

```
extern int register_framebuffer(struct fb_info *fb_info);
extern int unregister_framebuffer(struct fb_info *fb_info);
```

其中函数 register_framebuffer()的具体实现代码如下所示。

```
register_framebuffer(struct fb_info *fb_info)
{
        int ret;

        mutex_lock(&registration_lock);
        ret = do_register_framebuffer(fb_info);
        mutex_unlock(&registration_lock);

        return ret;
}
EXPORT_SYMBOL(register_framebuffer);
```

而函数 unregister_framebuffer()的具体实现代码如下所示。

```
*/
int
unregister_framebuffer(struct fb_info *fb_info)
{
        int ret;

        mutex_lock(&registration_lock);
        ret = do_unregister_framebuffer(fb_info);
        mutex_unlock(&registration_lock);

        return ret;
}
EXPORT_SYMBOL(unregister_framebuffer);
```

（2）定义如下所示的全局变量。

```
struct fb_info *registered_fb[FB_MAX] __read_mostly;
```

通过上述全局变量，在系统内可以随时获取需要的 fb_info。

（3）再看函数 fb_get_color_depth()，功能是获取颜色深度，如果是单色则深度为 1，否则深度为 red、blue、green 这 3 个分量的和。函数 fb_get_color_depth()的具体实现代码如下所示。

```
int fb_get_color_depth(struct fb_var_screeninfo *var,
                       struct fb_fix_screeninfo *fix)
{
        int depth = 0;
```

```

        if (fix->visual == FB_VISUAL_MONO01 ||
            fix->visual == FB_VISUAL_MONO10)
                depth = 1;
        else {
                if (var->green.length == var->blue.length &&
                    var->green.length == var->red.length &&
                    var->green.offset == var->blue.offset &&
                    var->green.offset == var->red.offset)
                        depth = var->green.length;
                else
                        depth = var->green.length + var->red.length +
                                var->blue.length;
        }

        return depth;
}
EXPORT_SYMBOL(fb_get_color_depth);
```

（4）函数 fb_get_buffer_offset()的功能是获取@buf 中符合@size 大小的空闲位置，具体实现代码如下所示。

```
char* fb_get_buffer_offset(struct fb_info *info, struct fb_pixmap *buf, u32 size)
{
        u32 align = buf->buf_align - 1, offset;
        char *addr = buf->addr;

        /* If IO mapped, we need to sync before access, no sharing of
         * the pixmap is done
         */
        if (buf->flags & FB_PIXMAP_IO) {
                if (info->fbops->fb_sync && (buf->flags & FB_PIXMAP_SYNC))
                        info->fbops->fb_sync(info);
                return addr;
        }
        /* See if we fit in the remaining pixmap space */
        offset = buf->offset + align;
        offset &= ~align;
        //如果剩余空间小于需要的大小，那么 fb_sync 后就可以使用@buffer 的所有空间
        if (offset + size > buf->size) {
                /* We do not fit. In order to be able to re-use the buffer,
                 * we must ensure no asynchronous DMA'ing or whatever operation
                 * is in progress, we sync for that.
                 */
                if (info->fbops->fb_sync && (buf->flags & FB_PIXMAP_SYNC))
                        info->fbops->fb_sync(info);
                offset = 0;
        }
        buf->offset = offset + size;
        addr += offset;

        return addr;
}
EXPORT_SYMBOL(fb_get_buffer_offset);
```

（5）函数 fb_set_logocmap()的功能是设置硬件的调色板颜色以及 info->cmap，具体实现代码如下所示。

```
static void fb_set_logocmap(struct fb_info *info,
                                        const struct linux_logo *logo)
{
        struct fb_cmap palette_cmap;
        u16 palette_green[16];
        u16 palette_blue[16];
        u16 palette_red[16];
        int i, j, n;
        const unsigned char *clut = logo->clut;

        palette_cmap.start = 0;
        palette_cmap.len = 16;
        palette_cmap.red = palette_red;
        palette_cmap.green = palette_green;
        palette_cmap.blue = palette_blue;
        palette_cmap.transp = NULL;

        for (i = 0; i < logo->clutsize; i += n) {
                n = logo->clutsize - i;
                /* palette_cmap provides space for only 16 colors at once */
                if (n > 16)
                        n = 16;
                 //之所以选 32 是因为 CLUT224 这种格式的 index 值为 32～255，即表示在 linux_logo->data
中只能找到 0 值，以及 32~255 之间的值
                palette_cmap.start = 32 + i;
                palette_cmap.len = n;
                for (j = 0; j < n; ++j) {
                        palette_cmap.red[j] = clut[0] << 8 | clut[0];
                        palette_cmap.green[j] = clut[1] << 8 | clut[1];
                        palette_cmap.blue[j] = clut[2] << 8 | clut[2];
                        clut += 3;
                }
                fb_set_cmap(&palette_cmap, info);
        }
}
```

在 Linux 系统中，支持如下几种常见的色彩模式。

```
#define FB_VISUAL_MONO01           0
#define FB_VISUAL_MONO10           1    /* Monochr. 1=White 0=Black */
#define FB_VISUAL_TRUECOLOR        2    /* True color    */
#define FB_VISUAL_PSEUDOCOLOR      3    /* Pseudo color (like atari) */
#define FB_VISUAL_DIRECTCOLOR      4    /* Direct color */
FB_VISUAL_MONO10 FB_VISUAL_MONO01：每个像素为黑或者白
FB_VISUAL_TRUECOLOR：真彩色，分为红、蓝、绿三基色
FB_VISUAL_PSEUDOCOLOR：伪彩色，采用索引颜色显示，需要根据颜色 index 查找 colormap，找到相应的
颜色值
FB_VISUAL_DIRECTORCOLOR：每个像素颜色也是由红、绿、蓝 3 种颜色组成，不过每个颜色都是索引值，具
体值需要查表
```

在结构 fb_cmap 中定义了具体的颜色表，具体代码如下所示。

```
struct fb_cmap {
     __u32 start;                          /*  第一个 entry，  没看出 start 的作用  */
```

```
    __u32 len;                  /* 每个颜色分量的长度 */
    __u16 *red;                 /* 红色分量 */
    __u16 *green;
    __u16 *blue;
    __u16 *transp;              /* 透明度，可以为空 */

};
```

在结构 linux_log 中定义了一个 Linux logo 的全部信息，具体代码如下所示。

```
struct linux_logo {
    int type;                   /* one of LINUX_LOGO_*，logo 的类型 */
    unsigned int width;         /* logo 的宽度*/
    unsigned int height;        /* logo 的高度*/
    unsigned int clutsize;      /* LINUX_LOGO_CLUT224 only，颜色查找表的尺寸 */
    const unsigned char *clut;  /* LINUX_LOGO_CLUT224 only，颜色查找表*/
    const unsigned char *data; /* logo 文件数据，对于 LINUX_LOGO_CLUT224，data 保存查找表的位置 */
};
```

（6）函数 fb_set_logo_truepalette()的功能是为 FB_VISUAL_PSEUDOCOLOR 彩色模式的 logo 生成一个调色板。此处是从 32 开始，因为 CLUT224 只支持 32～255 范围内的 index 值。函数 fb_set_logo_truepalette()的具体实现代码如下所示。

```
static void  fb_set_logo_truepalette(struct fb_info *info,
                                    const struct linux_logo *logo,
                                    u32 *palette)
{
        static const unsigned char mask[] = { 0,0x80,0xc0,0xe0,0xf0,0xf8,0xfc,0xfe,0xff };
        unsigned char redmask, greenmask, bluemask;
        int redshift, greenshift, blueshift;
        int i;
        const unsigned char *clut = logo->clut;

        /*
         * We have to create a temporary palette since console palette is only
         * 16 colors long.
         */
        /* Bug: Doesn't obey msb_right ... (who needs that?) */
        redmask   = mask[info->var.red.length   < 8 ? info->var.red.length   : 8];
        greenmask = mask[info->var.green.length < 8 ? info->var.green.length : 8];
        bluemask  = mask[info->var.blue.length  < 8 ? info->var.blue.length  : 8];
        redshift   = info->var.red.offset   - (8 - info->var.red.length);
        greenshift = info->var.green.offset - (8 - info->var.green.length);
        blueshift  = info->var.blue.offset  - (8 - info->var.blue.length);

        for ( i = 0; i < logo->clutsize; i++) {
                palette[i+32] = (safe_shift((clut[0] & redmask), redshift) |
                                 safe_shift((clut[1] & greenmask), greenshift) |
                                 safe_shift((clut[2] & bluemask), blueshift));
                clut += 3;
        }
}
```

（7）函数 fb_set_logo_directpalette()的功能是为 FB_VISUAL_DIRECTCOLOR 彩色模式生成一个调色板，此处只需生成 32 到 clutsize 的调色板即可。函数 fb_set_logo_directpalette()的具体实现代码如下所示。



```
static void fb_set_logo_directpalette(struct fb_info *info,
                                      const struct linux_logo *logo,
                                      u32 *palette)
{
        int redshift, greenshift, blueshift;
        int i;

        redshift = info->var.red.offset;
        greenshift = info->var.green.offset;
        blueshift = info->var.blue.offset;

        for (i = 32; i < 32 + logo->clutsize; i++)
                palette[i] = i << redshift | i << greenshift | i << blueshift;
}
```

（8）函数 fb_set_logo()的功能是实现@depth 数据和@logo 数据的转换。因为在 linux_logo->data 中保存的是 logo 的 data 数据，这对于 mono 或者 16 色的数据来说，linxu_logo->data 中的每个字节保存的是多个像素点的数据。此处函数 fb_set_logo()会根据颜色深度把 linux_logo->data 的数据转换到@dst 中，@dst 中的每个字节代表这一个像素索引。函数 fb_set_logo()的具体实现代码如下所示。

```
static void fb_set_logo(struct fb_info *info,
                        const struct linux_logo *logo, u8 *dst,
                        int depth)
{
        int i, j, k;
        const u8 *src = logo->data;
        u8 xor = (info->fix.visual == FB_VISUAL_MONO01) ? 0xff : 0;
        u8 fg = 1, d;

        switch (fb_get_color_depth(&info->var, &info->fix)) {
        case 1:
                fg = 1;
                break;
        case 2:
                fg = 3;
                break;
        default:
                fg = 7;
                break;
        }

        if (info->fix.visual == FB_VISUAL_MONO01 ||
            info->fix.visual == FB_VISUAL_MONO10)
                fg = ~((u8) (0xfff << info->var.green.length));

        switch (depth) {
        case 4:
                for (i = 0; i < logo->height; i++)
                        for (j = 0; j < logo->width; src++) {
                                *dst++ = *src >> 4;
                                j++;
                                if (j < logo->width) {
                                        *dst++ = *src & 0x0f;
```

```
                                                j++;
                                        }
                                }
                        break;
                case 1:
                        for (i = 0; i < logo->height; i++) {
                                for (j = 0; j < logo->width; src++) {
                                        d = *src ^ xor;
                                        for (k = 7; k >= 0; k--) {
                                                *dst++ = ((d >> k) & 1) ? fg : 0;
                                                j++;
                                        }
                                }
                        }
                        break;
                }
}
```

上述代码中涉及了结构体 logo_data，具体代码如下所示。

```
static struct logo_data {
        int depth;
        int needs_directpalette;
        int needs_truepalette;
        int needs_cmapreset;
        const struct linux_logo *logo;
} fb_logo __read_mostly;
```

（9）函数 fb_rotate_logo_ud()的功能是实现屏幕上下颠倒时的处理，函数 fb_rotate_logo_cw()的功能是实现顺时针旋转 90° 时的处理，函数 fb_rotate_logo_ccw()的功能是实现逆时针旋转 90° 时的处理。这 3 个函数的具体代码如下所示。

```
static void fb_rotate_logo_ud(const u8 *in, u8 *out, u32 width, u32 height)
{
        u32 size = width * height, i;

        out += size - 1;

        for (i = size; i--; )
                *out-- = *in++;
}
static void fb_rotate_logo_cw(const u8 *in, u8 *out, u32 width, u32 height)
{
        int i, j, h = height - 1;

        for (i = 0; i < height; i++)
                for (j = 0; j < width; j++)
                                out[height * j + h - i] = *in++;
}

static void fb_rotate_logo_ccw(const u8 *in, u8 *out, u32 width, u32 height)
{
        int i, j, w = width - 1;

```

```
        for (i = 0; i < height; i++)
                for (j = 0; j < width; j++)
                        out[height * (w - j) + i] = *in++;
}
```

（10）函数 fb_rotate_logo()的功能是显示@image 中的 logo 数据，其中参数@rotate 表示旋转方式。函数 fb_rotate_logo()的具体实现代码如下所示。

```
static void fb_rotate_logo(struct fb_info *info, u8 *dst,
                                struct fb_image *image, int rotate)
{
        u32 tmp;

        if (rotate == FB_ROTATE_UD) {
                fb_rotate_logo_ud(image->data, dst, image->width,
                                  image->height);
                image->dx = info->var.xres - image->width - image->dx;
                image->dy = info->var.yres - image->height - image->dy;
        } else if (rotate == FB_ROTATE_CW) {
                fb_rotate_logo_cw(image->data, dst, image->width,
                                  image->height);
                tmp = image->width;
                image->width = image->height;
                image->height = tmp;
                tmp = image->dy;
                image->dy = image->dx;
                image->dx = info->var.xres - image->width - tmp;
        } else if (rotate == FB_ROTATE_CCW) {
                fb_rotate_logo_ccw(image->data, dst, image->width,
                                   image->height);
                tmp = image->width;
                image->width = image->height;
                image->height = tmp;
                tmp = image->dx;
                image->dx = image->dy;
                image->dy = info->var.yres - image->height - tmp;
        }

        image->data = dst;
}
```

（11）函数 fb_show_logo_line()的功能是根据参数 logo 的内容构造一个 fb_image 结构体 image，用来描述最终要显示的第一个开机画面。函数 fb_show_logo_line()的具体实现代码如下所示。

```
static int fb_show_logo_line(struct fb_info *info, int rotate,
                                const struct linux_logo *logo, int y,
                                unsigned int n)
{
        u32 *palette = NULL, *saved_pseudo_palette = NULL;
        unsigned char *logo_new = NULL, *logo_rotate = NULL;
        struct fb_image image;

        /* Return if the frame buffer is not mapped or suspended */
        if (logo == NULL || info->state != FBINFO_STATE_RUNNING ||
```

```
			info->flags & FBINFO_MODULE)
				return 0;

		image.depth = 8;
		image.data = logo->data;

		if (fb_logo.needs_cmapreset)
				fb_set_logocmap(info, logo);

		if (fb_logo.needs_truepalette ||
			fb_logo.needs_directpalette) {
				palette = kmalloc(256 * 4, GFP_KERNEL);
				if (palette == NULL)
						return 0;

				if (fb_logo.needs_truepalette)
						fb_set_logo_truepalette(info, logo, palette);
				else
						fb_set_logo_directpalette(info, logo, palette);

				saved_pseudo_palette = info->pseudo_palette;
				info->pseudo_palette = palette;
		}

		if (fb_logo.depth <= 4) {
				logo_new = kmalloc(logo->width * logo->height, GFP_KERNEL);
				if (logo_new == NULL) {
						kfree(palette);
						if (saved_pseudo_palette)
								info->pseudo_palette = saved_pseudo_palette;
						return 0;
				}
				image.data = logo_new;
				fb_set_logo(info, logo, logo_new, fb_logo.depth);
		}

		image.dx = 0;
		image.dy = y;
		image.width = logo->width;
		image.height = logo->height;

		if (rotate) {
				logo_rotate = kmalloc(logo->width *
									logo->height, GFP_KERNEL);
				if (logo_rotate)
						fb_rotate_logo(info, logo_rotate, &image, rotate);
		}

		fb_do_show_logo(info, &image, rotate, n);

		kfree(palette);
```

```
        if (saved_pseudo_palette != NULL)
                info->pseudo_palette = saved_pseudo_palette;
        kfree(logo_new);
        kfree(logo_rotate);
        return logo->height;
}
```

（12）函数 fb_do_show_logo()的功能是实现真正执行渲染第一个开机画面的操作，具体实现代码如下所示。

```
static void fb_do_show_logo(struct fb_info *info, struct fb_image *image,
                                int rotate, unsigned int num)
{
        unsigned int x;

        if (rotate == FB_ROTATE_UR) {
                for (x = 0;
                     x < num && image->dx + image->width <= info->var.xres;
                     x++) {
                        info->fbops->fb_imageblit(info, image);
                        image->dx += image->width + 8;
                }
        } else if (rotate == FB_ROTATE_UD) {
                for (x = 0; x < num && image->dx >= 0; x++) {
                        info->fbops->fb_imageblit(info, image);
                        image->dx -= image->width + 8;
                }
        } else if (rotate == FB_ROTATE_CW) {
                for (x = 0;
                     x < num && image->dy + image->height <= info->var.yres;
                     x++) {
                        info->fbops->fb_imageblit(info, image);
                        image->dy += image->height + 8;
                }
        } else if (rotate == FB_ROTATE_CCW) {
                for (x = 0; x < num && image->dy >= 0; x++) {
                        info->fbops->fb_imageblit(info, image);
                        image->dy -= image->height + 8;
                }
        }
}
```

在上述代码中，参数 rotate 表示屏幕的当前旋转方向。根据屏幕旋转方向不同，第一个开机画面的渲染方式也有所不同。例如，当屏幕上下颠倒时（FB_ROTATE_UD），第一个开机画面的左右顺序就刚好调换过来，这时就需要从右到左来渲染。其他 3 个方向 FB_ROTATE_UR、FB_ROTATE_CW 和 FB_ROTATE_CCW 分别表示没有旋转、顺时针旋转 90° 和逆时针旋转 90° 。参数 info 用来描述要渲染的帧缓冲区硬件设备，它的成员变量 fbops 指向了一系列回调函数，用来操作帧缓冲区硬件设备，其中，回调函数 fb_imageblit()用来在指定的帧缓冲区硬件设备渲染指定的图像。

（13）函数 fb_append_extra_logo()的功能是将给定的 logo 设置到全局 extend logo 数组 fb_logo_ex 中，具体实现代码如下所示。

```
void fb_append_extra_logo(const struct linux_logo *logo, unsigned int n)
{
```

```
        if (!n || fb_logo_ex_num == FB_LOGO_EX_NUM_MAX)
                return;

        fb_logo_ex[fb_logo_ex_num].logo = logo;
        fb_logo_ex[fb_logo_ex_num].n = n;
        fb_logo_ex_num++;
}
```

（14）函数 fb_prepare_extra_logos()的功能是计算 height 和 fb_logo_ex_num 的值，其中 height 是 logo 和有效 extend logo 的高度和，而 fb_log_ex_num 是有效 extend logo 的最大索引。函数 fb_prepare_extra_logos()的具体实现代码如下所示。

```
static int fb_prepare_extra_logos(struct fb_info *info, unsigned int height,
                                          unsigned int yres)
{
        unsigned int i;

        /* FIXME: logo_ex supports only truecolor fb. */
        if (info->fix.visual != FB_VISUAL_TRUECOLOR)
                fb_logo_ex_num = 0;

        for (i = 0; i < fb_logo_ex_num; i++) {
                if (fb_logo_ex[i].logo->type != fb_logo.logo->type) {
                        fb_logo_ex[i].logo = NULL;
                        continue;
                }
                height += fb_logo_ex[i].logo->height;
                if (height > yres) {
                        height -= fb_logo_ex[i].logo->height;
                        fb_logo_ex_num = i;
                        break;
                }
        }
        return height;
}
```

（15）函数 fb_show_extra_logos()的功能是显示保存在 fb_logo_ex 中的 extend logo，其中参数 y 表示这个 extend logo 要在屏幕显示的位置。函数 fb_show_extra_logos()的具体实现代码如下所示。

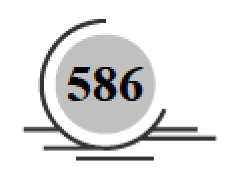
```
static int fb_show_extra_logos(struct fb_info *info, int y, int rotate)
{
        unsigned int i;

        for (i = 0; i < fb_logo_ex_num; i++)
                y += fb_show_logo_line(info, rotate,
                                        fb_logo_ex[i].logo, y, fb_logo_ex[i].n);

        return y;
}
```

（16）函数 fb_prepare_logo()的功能是根据 fb_info 获取颜色 depth，并根据 depth 获取合适的 logo，通过 fb_find_logo 根据 depth 找到适合的 logo，最后根据获得的 logo 类型计算 logo 的 depth。函数 fb_prepare_logo()的具体实现代码如下所示。

```
int fb_prepare_logo(struct fb_info *info, int rotate)
{
```



```
	int depth = fb_get_color_depth(&info->var, &info->fix);
	unsigned int yres;

	memset(&fb_logo, 0, sizeof(struct logo_data));

	if (info->flags & FBINFO_MISC_TILEBLITTING ||
	    info->flags & FBINFO_MODULE)
		return 0;

	if (info->fix.visual == FB_VISUAL_DIRECTCOLOR) {
		depth = info->var.blue.length;
		if (info->var.red.length < depth)
			depth = info->var.red.length;
		if (info->var.green.length < depth)
			depth = info->var.green.length;
	}

	if (info->fix.visual == FB_VISUAL_STATIC_PSEUDOCOLOR && depth > 4) {
		/* assume console colormap */
		depth = 4;
	}

	/* Return if no suitable logo was found */
	fb_logo.logo = fb_find_logo(depth);

	if (!fb_logo.logo) {
		return 0;
	}

	if (rotate == FB_ROTATE_UR || rotate == FB_ROTATE_UD)
		yres = info->var.yres;
	else
		yres = info->var.xres;

	if (fb_logo.logo->height > yres) {
		fb_logo.logo = NULL;
		return 0;
	}

	/* What depth we asked for might be different from what we get */
	if (fb_logo.logo->type == LINUX_LOGO_CLUT224)
		fb_logo.depth = 8;
	else if (fb_logo.logo->type == LINUX_LOGO_VGA16)
		fb_logo.depth = 4;
	else
		fb_logo.depth = 1;


	if (fb_logo.depth > 4 && depth > 4) {
		switch (info->fix.visual) {
		case FB_VISUAL_TRUECOLOR:
```

```
				fb_logo.needs_truepalette = 1;
				break;
			case FB_VISUAL_DIRECTCOLOR:
				fb_logo.needs_directpalette = 1;
				fb_logo.needs_cmapreset = 1;
				break;
			case FB_VISUAL_PSEUDOCOLOR:
				fb_logo.needs_cmapreset = 1;
				break;
			}
	}

	return fb_prepare_extra_logos(info, fb_logo.logo->height, yres);
}
```

（17）函数 fb_show_logo()的功能在显示 logo 后通过 fb_show_logo_line 返回 logo 占用的 vertical height（垂直高度），然后在 logo 下显示 extra logo，此处传入的参数 y 就是 logo 的 height（高度）。函数 fb_show_logo()的具体实现代码如下所示。

```
int fb_show_logo(struct fb_info *info, int rotate)
{
	int y;

	y = fb_show_logo_line(info, rotate, fb_logo.logo, 0,
			      num_online_cpus());
	y = fb_show_extra_logos(info, y, rotate);

	return y;
}
```

（18）函数 fb_read()的功能是读取设备文件中的一段数据。对于 FrameBuffer 驱动系统来说，这些被读取数据保存在虚拟地址 info->screen_base 中。info->screen_base 是 framebuffer mem 的虚拟地址，info->fix.smem_start 是 framebuffer mem 的物理地址，通常驱动访问的是 info->screen_base。函数 fb_read()的具体实现代码如下所示。

```
static ssize_t
fb_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
	unsigned long p = *ppos;
	struct fb_info *info = file_fb_info(file);
	u8 *buffer, *dst;
	u8 __iomem *src;
	int c, cnt = 0, err = 0;
	unsigned long total_size;

	if (!info || ! info->screen_base)
		return -ENODEV;

	if (info->state != FBINFO_STATE_RUNNING)
		return -EPERM;

	if (info->fbops->fb_read)
		return info->fbops->fb_read(info, buf, count, ppos);
```

```

	total_size = info->screen_size;

	if (total_size == 0)
		total_size = info->fix.smem_len;

	if (p >= total_size)
		return 0;

	if (count >= total_size)
		count = total_size;

	if (count + p > total_size)
		count = total_size - p;

	buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count,
			 GFP_KERNEL);
	if (!buffer)
		return -ENOMEM;

	src = (u8 __iomem *) (info->screen_base + p);

	if (info->fbops->fb_sync)
		info->fbops->fb_sync(info);

	while (count) {
		c  = (count > PAGE_SIZE) ? PAGE_SIZE : count;
		dst = buffer;
		fb_memcpy_fromfb(dst, src, c);
		dst += c;
		src += c;

		if (copy_to_user(buf, buffer, c)) {
			err = -EFAULT;
			break;
		}
		*ppos += c;
		buf += c;
		cnt += c;
		count -= c;
	}

	kfree(buffer);

	return (err) ? err : cnt;
}
```

在上述代码中，通过 fb_readl 和 fb_readb 来读取 info->screen_base 的内容，并复制到参数 buf 中。

（19）函数 fb_pan_display()是 FBIOPAN_DISPLAY 的实现，功能是通过参数 var 的 xoffset 和 yoffset 实现屏幕内容的平滑移动。函数 fb_pan_display()的具体实现代码如下所示。

```
int
fb_pan_display(struct fb_info *info, struct fb_var_screeninfo *var)
```

```
{
	struct fb_fix_screeninfo *fix = &info->fix;
	unsigned int yres = info->var.yres;
	int err = 0;

	if (var->yoffset > 0) {
		if (var->vmode & FB_VMODE_YWRAP) {
			if (!fix->ywrapstep || (var->yoffset % fix->ywrapstep))
				err = -EINVAL;
			else
				yres = 0;
		} else if (!fix->ypanstep || (var->yoffset % fix->ypanstep))
			err = -EINVAL;
	}

	if (var->xoffset > 0 && (!fix->xpanstep ||
				 (var->xoffset % fix->xpanstep)))
		err = -EINVAL;

	if (err || !info->fbops->fb_pan_display ||
	    var->yoffset > info->var.yres_virtual - yres ||
	    var->xoffset > info->var.xres_virtual - info->var.xres)
		return -EINVAL;

	if ((err = info->fbops->fb_pan_display(var, info)))
		return err;
	info->var.xoffset = var->xoffset;
	info->var.yoffset = var->yoffset;
	if (var->vmode & FB_VMODE_YWRAP)
		info->var.vmode |= FB_VMODE_YWRAP;
	else
		info->var.vmode &= ~FB_VMODE_YWRAP;
	return 0;
}
EXPORT_SYMBOL(fb_pan_display);
```

（20）函数 fb_set_var()的功能是实现显示模式和可变参数的设置。此函数在不同的显示驱动中的具体名称也不一样，但是基本上的功能都是完成对模式和可变参数的控制。函数 fb_set_var()的具体实现代码如下所示。

```
int
fb_set_var(struct fb_info *info, struct fb_var_screeninfo *var)
{
	int flags = info->flags;
	int ret = 0;

	if (var->activate & FB_ACTIVATE_INV_MODE) {
		struct fb_videomode mode1, mode2;

	//转换 var 和当前 fb_info->var 到 viewmode，如果@var 对应的 viewmode 不是当前正在使用的viewmode，那么调用 notifier()函数，并从 info->modelist 中删除所有匹配的 viewmode
		fb_var_to_videomode(&mode1, var);
		fb_var_to_videomode(&mode2, &info->var);
```

```
				/* make sure we don't delete the videomode of current var */
				ret = fb_mode_is_equal(&mode1, &mode2);

				if (!ret) {
					struct fb_event event;

					event.info = info;
					event.data = &mode1;
					ret = fb_notifier_call_chain(FB_EVENT_MODE_DELETE, &event);
				}

				if (!ret)
					fb_delete_videomode(&mode1, &info->modelist);


				ret = (ret) ? -EINVAL : 0;
				goto done;
		}
		//如果 var->active 是 FB_ACTIVE_NOW， 那么激活给定的@var
		if ((var->activate & FB_ACTIVATE_FORCE) ||
			memcmp(&info->var, var, sizeof(struct fb_var_screeninfo))) {
				u32 activate = var->activate;

				/* When using FOURCC mode, make sure the red, green, blue and
				 * transp fields are set to 0.
				 */
				if ((info->fix.capabilities & FB_CAP_FOURCC) &&
					var->grayscale > 1) {
						if (var->red.offset		|| var->green.offset	||
							var->blue.offset	|| var->transp.offset	||
							var->red.length		|| var->green.length	||
							var->blue.length	|| var->transp.length	||
							var->red.msb_right	|| var->green.msb_right ||
							var->blue.msb_right || var->transp.msb_right)
								return -EINVAL;
				}

				if (!info->fbops->fb_check_var) {
						*var = info->var;
						goto done;
				}

				ret = info->fbops->fb_check_var(var, info);

				if (ret)
						goto done;


				if ((var->activate & FB_ACTIVATE_MASK) == FB_ACTIVATE_NOW) {
						struct fb_var_screeninfo old_var;
						struct fb_videomode mode;
```

```
				if (info->fbops->fb_get_caps) {
					ret = fb_check_caps(info, var, activate);

					if (ret)
						goto done;
				}
	//设置 info->var 为@var，并且调用 fb_set_par()函数设置新的 framebuffer 参数，改变操作模式
				old_var = info->var;
				info->var = *var;

				if (info->fbops->fb_set_par) {
					ret = info->fbops->fb_set_par(info);

					if (ret) {
						info->var = old_var;
						printk(KERN_WARNING "detected "
							"fb_set_par error, "
							"error code: %d\n", ret);
						goto done;
					}
				}
				//在设置新的 framebuffer 后需要调用 fb_pan_display()函数来更新 pan display，
fb_pan_display()函数需要特定的 framebuffer 实现
				fb_pan_display(info, &info->var);
				fb_set_cmap(&info->cmap, info);
				//把 var 对应的 videomode 加入到 modelist 中
				fb_var_to_videomode(&mode, &info->var);

				if (info->modelist.prev && info->modelist.next &&
				    !list_empty(&info->modelist))
					ret = fb_add_videomode(&mode, &info->modelist);
				//下面一段是广播 framebuffer 事件
				if (!ret && (flags & FBINFO_MISC_USEREVENT)) {
					struct fb_event event;
					int evnt = (activate & FB_ACTIVATE_ALL) ?
						FB_EVENT_MODE_CHANGE_ALL :
						FB_EVENT_MODE_CHANGE;

					info->flags &= ~FBINFO_MISC_USEREVENT;
					event.info = info;
					event.data = &mode;
					fb_notifier_call_chain(evnt, &event);
				}
			}
		}

	done:
		return ret;
}
EXPORT_SYMBOL(fb_set_var);
```

（21）函数 fb_blank()的功能是根据参数 blank 指定 blank 的类型，并重新点亮 display 显示。blank 的类型有 POWERDOWN、NORMAL、HSYNC_SUSPEND、VSYNC_SUSPEND，调用顺序是 info->fbops->fb_blank。函数 fb_blank()的具体实现代码如下所示。

```
int
 fb_blank (struct fb_info *info, int blank)
{
        struct fb_event event;
        int ret = -EINVAL, early_ret;

        if (blank > FB_BLANK_POWERDOWN)
                blank = FB_BLANK_POWERDOWN;

        event.info = info;
        event.data = &blank;

        early_ret = fb_notifier_call_chain(FB_EARLY_EVENT_BLANK, &event);

        if (info->fbops->fb_blank)
                ret = info->fbops->fb_blank(blank, info);

        if (!ret)
                fb_notifier_call_chain(FB_EVENT_BLANK, &event);
        else {
                /*
                 * if fb_blank is failed then revert effects of
                 * the early blank event.
                 */
                if (!early_ret)
                        fb_notifier_call_chain(FB_R_EARLY_EVENT_BLANK, &event);
        }

        return ret;
}
EXPORT_SYMBOL(fb_blank);
```

（22）函数 do_fb_ioctl()的功能是根据 case 语句执行对应的 ioctl 处理函数，具体实现代码如下所示。

```
static long do_fb_ioctl(struct fb_info *info, unsigned int cmd,
                        unsigned long arg)
{
        struct fb_ops *fb;
        struct fb_var_screeninfo var;
        struct fb_fix_screeninfo fix;
        struct fb_con2fbmap con2fb;
        struct fb_cmap cmap_from;
        struct fb_cmap_user cmap;
        struct fb_event event;
        void __user *argp = (void __user *)arg;
        long ret = 0;

        switch (cmd) {
        case FBIOGET_VSCREENINFO:
```

```
			if (!lock_fb_info(info))
				return -ENODEV;
			var = info->var;
			unlock_fb_info(info);

			ret = copy_to_user(argp, &var, sizeof(var)) ? -EFAULT : 0;
			break;
		case FBIOPUT_VSCREENINFO:
			if (copy_from_user(&var, argp, sizeof(var)))
				return -EFAULT;
			if (!lock_fb_info(info))
				return -ENODEV;
			console_lock();
			info->flags |= FBINFO_MISC_USEREVENT;
			ret = fb_set_var(info, &var);
			info->flags &= ~FBINFO_MISC_USEREVENT;
			console_unlock();
			unlock_fb_info(info);
			if (!ret && copy_to_user(argp, &var, sizeof(var)))
				ret = -EFAULT;
			break;
		case FBIOGET_FSCREENINFO:
			if (!lock_fb_info(info))
				return -ENODEV;
			fix = info->fix;
			unlock_fb_info(info);

			ret = copy_to_user(argp, &fix, sizeof(fix)) ? -EFAULT : 0;
			break;
		case FBIOPUTCMAP:
			if (copy_from_user(&cmap, argp, sizeof(cmap)))
				return -EFAULT;
			ret = fb_set_user_cmap(&cmap, info);
			break;
		case FBIOGETCMAP:
			if (copy_from_user(&cmap, argp, sizeof(cmap)))
				return -EFAULT;
			if (!lock_fb_info(info))
				return -ENODEV;
			cmap_from = info->cmap;
			unlock_fb_info(info);
			ret = fb_cmap_to_user(&cmap_from, &cmap);
			break;
		case FBIOPAN_DISPLAY:
			if (copy_from_user(&var, argp, sizeof(var)))
				return -EFAULT;
			if (!lock_fb_info(info))
				return -ENODEV;
			console_lock();
			ret = fb_pan_display(info, &var);
			console_unlock();
```

```
		unlock_fb_info(info);
		if (ret == 0 && copy_to_user(argp, &var, sizeof(var)))
			return -EFAULT;
		break;
	case FBIO_CURSOR:
		ret = -EINVAL;
		break;
	case FBIOGET_CON2FBMAP:
		if (copy_from_user(&con2fb, argp, sizeof(con2fb)))
			return -EFAULT;
		if (con2fb.console < 1 || con2fb.console > MAX_NR_CONSOLES)
			return -EINVAL;
		con2fb.framebuffer = -1;
		event.data = &con2fb;
		if (!lock_fb_info(info))
			return -ENODEV;
		event.info = info;
		fb_notifier_call_chain(FB_EVENT_GET_CONSOLE_MAP, &event);
		unlock_fb_info(info);
		ret = copy_to_user(argp, &con2fb, sizeof(con2fb)) ? -EFAULT : 0;
		break;
	case FBIOPUT_CON2FBMAP:
		if (copy_from_user(&con2fb, argp, sizeof(con2fb)))
			return -EFAULT;
		if (con2fb.console < 1 || con2fb.console > MAX_NR_CONSOLES)
			return -EINVAL;
		if (con2fb.framebuffer < 0 || con2fb.framebuffer >= FB_MAX)
			return -EINVAL;
		if (!registered_fb[con2fb.framebuffer])
			request_module("fb%d", con2fb.framebuffer);
		if (!registered_fb[con2fb.framebuffer]) {
			ret = -EINVAL;
			break;
		}
		event.data = &con2fb;
		if (!lock_fb_info(info))
			return -ENODEV;
		console_lock();
		event.info = info;
		ret = fb_notifier_call_chain(FB_EVENT_SET_CONSOLE_MAP, &event);
		console_unlock();
		unlock_fb_info(info);
		break;
	case FBIOBLANK:
		if (!lock_fb_info(info))
			return -ENODEV;
		console_lock();
		info->flags |= FBINFO_MISC_USEREVENT;
		ret = fb_blank(info, arg);
		info->flags &= ~FBINFO_MISC_USEREVENT;
		console_unlock();
```

```
			unlock_fb_info(info);
			break;
		default:
			if (!lock_fb_info(info))
				return -ENODEV;
			fb = info->fbops;
			if (fb->fb_ioctl)
				ret = fb->fb_ioctl(info, cmd, arg);
			else
				ret = -ENOTTY;
			unlock_fb_info(info);
		}
		return ret;
}
```

（23）函数 fb_mmap()的功能是将 FrameBuffer 的物理内存映射到进程的虚拟地址空间中，具体实现代码如下所示。

```
fb_mmap(struct file *file, struct vm_area_struct * vma)
{
		struct fb_info *info = file_fb_info(file);
		struct fb_ops *fb;
		unsigned long mmio_pgoff;
		unsigned long start;
		u32 len;

		if (!info)
			return -ENODEV;
		fb = info->fbops;
		if (!fb)
			return -ENODEV;
		mutex_lock(&info->mm_lock);
		if (fb->fb_mmap) {
			int res;
			res = fb->fb_mmap(info, vma);
			mutex_unlock(&info->mm_lock);
			return res;
		}

		/*
		 * Ugh. This can be either the frame buffer mapping, or
		 * if pgoff points past it, the mmio mapping
		 */
		start = info->fix.smem_start;
		len = info->fix.smem_len;
		mmio_pgoff = PAGE_ALIGN((start & ~PAGE_MASK) + len) >> PAGE_SHIFT;
		if (vma->vm_pgoff >= mmio_pgoff) {
			if (info->var.accel_flags) {
				mutex_unlock(&info->mm_lock);
				return -EINVAL;
			}

```

```
			vma->vm_pgoff -= mmio_pgoff;
			start = info->fix.mmio_start;
			len = info->fix.mmio_len;
		}
		mutex_unlock(&info->mm_lock);

		vma->vm_page_prot = vm_get_page_prot(vma->vm_flags);
		fb_pgprotect(file, vma, start);

		return vm_iomap_memory(vma, start, len);
}
```

（24）函数 do_register_framebuffer()的功能是为 FrameBuffer 驱动提供注册一个 FrameBuffer 设备的接口，通过此函数会把参数 fb_info 添加到 registered_fb 中。函数 do_register_framebuffer()的具体实现代码如下所示。

```
static int do_register_framebuffer(struct fb_info *fb_info)
{
		int i;
		struct fb_event event;
		struct fb_videomode mode;

		if (fb_check_foreignness(fb_info))
			return -ENOSYS;

		do_remove_conflicting_framebuffers(fb_info->apertures, fb_info->fix.id,
						fb_is_primary_device(fb_info));

		if (num_registered_fb == FB_MAX)
			return -ENXIO;

		num_registered_fb++;
		for (i = 0 ; i < FB_MAX; i++)
			if (!registered_fb[i])
				break;
		fb_info->node = i;
		atomic_set(&fb_info->count, 1);
		mutex_init(&fb_info->lock);
		mutex_init(&fb_info->mm_lock);
		//为 FrameBuffer 设备创建 class device name
		fb_info->dev = device_create(fb_class, fb_info->device,
					MKDEV(FB_MAJOR, i), NULL, "fb%d", i);
		if (IS_ERR(fb_info->dev)) {
			/* Not fatal */
			printk(KERN_WARNING "Unable to create device for framebuffer %d; errno = %ld\n", i, PTR_ERR(fb_info->dev));
			fb_info->dev = NULL;
		} else
		// fb_init_device 创建 FrameBuffer 的 attr 文件
			fb_init_device(fb_info);
		if (fb_info->pixmap.addr == NULL) {
			fb_info->pixmap.addr = kmalloc(FBPIXMAPSIZE, GFP_KERNEL);
```

```
			if (fb_info->pixmap.addr) {
				fb_info->pixmap.size = FBPIXMAPSIZE;
				fb_info->pixmap.buf_align = 1;
				fb_info->pixmap.scan_align = 1;
				fb_info->pixmap.access_align = 32;
				fb_info->pixmap.flags = FB_PIXMAP_DEFAULT;
			}
		}
		fb_info->pixmap.offset = 0;

		if (!fb_info->pixmap.blit_x)
			fb_info->pixmap.blit_x = ~(u32)0;

		if (!fb_info->pixmap.blit_y)
			fb_info->pixmap.blit_y = ~(u32)0;

		if (!fb_info->modelist.prev || !fb_info->modelist.next)
			INIT_LIST_HEAD(&fb_info->modelist);

		if (fb_info->skip_vt_switch)
			pm_vt_switch_required(fb_info->dev, false);
		else
			pm_vt_switch_required(fb_info->dev, true);
		//转换 fb_info->var 为 videomode，然后把 videomode 加入到 modelist 中
		fb_var_to_videomode(&mode, &fb_info->var);
		fb_add_videomode(&mode, &fb_info->modelist);
		registered_fb[i] = fb_info;		//把 fb_info 添加到 registered_fb 数组中

		event.info = fb_info;
		if (!lock_fb_info(fb_info))
			return -ENODEV;
		console_lock();
		fb_notifier_call_chain(FB_EVENT_FB_REGISTERED, &event);
		console_unlock();
		unlock_fb_info(fb_info);
		return 0;
}
```

（25）函数 do_unregister_framebuffer()是函数 do_register_framebuffer()的反操作过程，功能是为 FrameBuffer 驱动提供注销一个 FrameBuffer 设备的接口。函数 do_unregister_framebuffer()的具体实现代码如下所示。

```
static int do_unregister_framebuffer(struct fb_info *fb_info)
{
		struct fb_event event;
		int i, ret = 0;

		i = fb_info->node;
		if (i < 0 || i >= FB_MAX || registered_fb[i] != fb_info)
			return -EINVAL;

		if (!lock_fb_info(fb_info))
			return -ENODEV;
```

```
	console_lock();
	event.info = fb_info;
	ret = fb_notifier_call_chain(FB_EVENT_FB_UNBIND, &event);
	console_unlock();
	unlock_fb_info(fb_info);

	if (ret)
		return -EINVAL;

	pm_vt_switch_unregister(fb_info->dev);

	unlink_framebuffer(fb_info);
	if (fb_info->pixmap.addr &&
	    (fb_info->pixmap.flags & FB_PIXMAP_DEFAULT))
		kfree(fb_info->pixmap.addr);
	fb_destroy_modelist(&fb_info->modelist);
	registered_fb[i] = NULL;
	num_registered_fb--;
	fb_cleanup_device(fb_info);
	event.info = fb_info;
	console_lock();
	fb_notifier_call_chain(FB_EVENT_FB_UNREGISTERED, &event);
	console_unlock();

	/* this may free fb info */
	put_fb_info(fb_info);
	return 0;
}
```

（26）函数 fb_new_modelist()的功能是逐一测试 info->modelist 中的每一个 mode，从 modelist 中删除无效的 mode 节点。函数 fb_new_modelist()的具体实现代码如下所示。

```
int fb_new_modelist(struct fb_info *info)
{
	struct fb_event event;
	struct fb_var_screeninfo var = info->var;
	struct list_head *pos, *n;
	struct fb_modelist *modelist;
	struct fb_videomode *m, mode;
	int err = 1;

	list_for_each_safe(pos, n, &info->modelist) {
		modelist = list_entry(pos, struct fb_modelist, list);
		m = &modelist->mode;
		fb_videomode_to_var(&var, m);
		var.activate = FB_ACTIVATE_TEST;
		err = fb_set_var(info, &var);
		fb_var_to_videomode(&mode, &var);
		if (err || !fb_mode_is_equal(m, &mode)) {
			list_del(pos);
			kfree(pos);
		}
	}
```

```

        err = 1;

        if (!list_empty(&info->modelist)) {
                event.info = info;
                err = fb_notifier_call_chain(FB_EVENT_NEW_MODELIST, &event);
        }

        return err;
}
```

（27）函数 fb_get_options()的功能是从 Linux 内核的参数 cmd 中提取和 FrameBuffer 相关的选项，具体实现代码如下所示。

```
int fb_get_options(const char *name, char **option)
{
        char *opt, *options = NULL;
        int retval = 0;
        int name_len = strlen(name), i;

        if (name_len && ofonly && strncmp(name, "offb", 4))
                retval = 1;

        if (name_len && !retval) {
                for (i = 0; i < FB_MAX; i++) {
                        if (video_options[i] == NULL)
                                continue;
                        if (!video_options[i][0])
                                continue;
                        opt = video_options[i];
                        if (!strncmp(name, opt, name_len) &&
                            opt[name_len] == ':')
                                options = opt + name_len + 1;
                }
        }
        if (options && !strncmp(options, "off", 3))
                retval = 1;

        if (option)
                *option = options;

        return retval;
}
EXPORT_SYMBOL(fb_get_options);
```

到此为止，Android 系统中显示驱动 FrameBuffer 内核层的实现源码分析完毕。

18.3 硬件抽象层详解

在 Eclair 及其后面版本中，Gralloc 模块是显示部分的硬件抽象层。因为 Gralloc 模块是灵活多变的，所



以移植方式也多种多样，具体来说有如下两种情况。

（1）如果继续使用 Android 中已经实现的 Gralloc 模块，就可以继续使用标准的 FrameBuffer 驱动程序，此时只需要移植 FrameBuffer 的内容即可。

（2）如果想自己实现特定的 Gralloc 模块，则此模块就是当前系统的显示设备和 Android 接口，此时显示设备可以是各种类型的驱动程序。

注意： 如果想对一个标准的 FrameBuffer 驱动程序实现优化改动，需要额外增加一些 ioctl 命令来获取额外的控制，例如可以通过 Android 的 pmem 驱动程序实现获取加速效果。

下面将详细分析 Android 系统中显示驱动硬件抽象层的具体实现过程。

18.3.1　Gralloc 模块的头文件

Gralloc 模块的头文件在文件 hardware/libhardware/include/hardware/gralloc.h 中定义，接下来将详细讲解此文件的实现流程。

（1）定义子设备和模块的名称，对应代码如下所示。

```
#define GRALLOC_HARDWARE_MODULE_ID "gralloc"
#define GRALLOC_HARDWARE_FB0 "fb0"
#define GRALLOC_HARDWARE_GPU0 "gpu0"
```

其中 gralloc 是硬件模块的名称，fb0 是 FrameBuffer 设备，gpu0 是图形处理单元设备。

（2）通过扩展定义 gralloc_module_t 实现 Gralloc 硬件模块，对应代码如下所示。

```
typedef struct gralloc_module_t {
    struct hw_module_t common;
int (*registerBuffer)(struct gralloc_module_t const* module,
                buffer_handle_t handle);
 int (*unregisterBuffer)(struct gralloc_module_t const* module,
                buffer_handle_t handle);
 int (*lock)(struct gralloc_module_t const* module,
                buffer_handle_t handle, int usage,
                int l, int t, int w, int h,
                void** vaddr);
int (*unlock)(struct gralloc_module_t const* module,
                buffer_handle_t handle);
 int (*perform)(struct gralloc_module_t const* module,
                int operation, ... );
 void* reserved_proc[7];
} gralloc_module_t;
```

上述 gralloc_module_t 是此头文件的核心，各个函数指针的具体说明如下所示。

☑　registerBuffer：在 alloc_device_t::alloc 前调用。

☑　lock：用于访问特定的缓冲区，在调用此接口时硬件设备需要结束渲染或完成同步处理。

☑　unlock：在所用 buffer 改变之后被调用。

☑　perform：用于未来某个用途所用。

（3）定义函数 gralloc_open()，用于打开 gralloc 的接口，此函数的实现代码如下所示。

```
static inline int gralloc_open(const struct hw_module_t* module,
            struct alloc_device_t** device) {
    return module->methods->open(module,
                GRALLOC_HARDWARE_GPU0, (struct hw_device_t**)device);
}
```

（4）定义函数 framebuffer_open()，用于打开 FrameBuffer 的接口，此函数的实现代码如下所示。

```
static inline int framebuffer_open(const struct hw_module_t* module,
            struct framebuffer_device_t** device) {
    return module->methods->open(module,
                GRALLOC_HARDWARE_FB0, (struct hw_device_t**)device);
}
```

（5）定义函数 framebuffer_open()，用于关闭 FrameBuffer 的接口，此函数的实现代码如下所示。

```
static inline int framebuffer_close(struct framebuffer_device_t* device) {
    return device->common.close(&device->common);
}
```

（6）定义函数 gralloc_close()，用于关闭 gralloc 的接口，此函数的实现代码如下所示。

```
static inline int gralloc_close(struct alloc_device_t* device) {
    return device->common.close(&device->common);
}
```

（7）设备 GRALLOC_HARDWARE_GPU0 对应的设备是结构体 alloc_device_t，设备 GRALLOC_HARDWARE_FB0 对应的设备是结构体 framebuffer_device_t，这两个结构体的具体定义代码如下所示。

```
typedef struct alloc_device_t {
    struct hw_device_t common;
 int (*alloc)(struct alloc_device_t* dev,                //以宽、高、颜色格式为参数来分配
            int w, int h, int format, int usage,
            buffer_handle_t* handle, int* stride);
int (*free)(struct alloc_device_t* dev,
            buffer_handle_t handle);

} alloc_device_t;

typedef struct framebuffer_device_t {
    struct hw_device_t common;
    const uint32_t  flags;
 const uint32_t  width;                                 //宽
    const uint32_t  height;                            //高
  const int        stride;                             //每行内容
const int        format;                               //颜色格式
const float      xdpi;                                 //X 方向像素密度
    const float      ydpi;                             //Y 方向像素密度
    const float      fps;                              //频率
    const int        minSwapInterval;
    const int        maxSwapInterval;
    int reserved[8];
    int (*setSwapInterval)(struct framebuffer_device_t* window,
            int interval);
    int (*setUpdateRect)(struct framebuffer_device_t* window,
            int left, int top, int width, int height);

    int (*post)(struct framebuffer_device_t* dev, buffer_handle_t buffer);
    int (*compositionComplete)(struct framebuffer_device_t* dev);
    void* reserved_proc[8];
} framebuffer_device_t;
```

18.3.2　硬件帧缓冲区

在 Android 系统中，Gralloc 模块由 gralloc_module_t 模块、alloc_device_t 设备和 framebuffer_device_t 设备 3 个结构体来描述，里面的函数指针具有非常重要的作用。Gralloc 模块是由 UI 库中的文件 frameworks/base/include/ui/FramebufferNativeWindow.cpp 调用的。

在文件 FramebufferNativeWindow.cpp 中定义了类 FramebufferNativeWindow，此类继承了 android_native_buffer_t，这是对上层的接口，表示一个本地窗口。

文件 FramebufferNativeWindow.cpp 的核心是构造函数 FramebufferNativeWindow()，具体实现代码如下所示。

```
FramebufferNativeWindow::FramebufferNativeWindow():BASE(),fbDev(0), grDev(0), mUpdateOnDemand(false){
    hw_module_t const* module;
    if (hw_get_module(GRALLOC_HARDWARE_MODULE_ID, &module) == 0) {
        int stride;
        int err;
        err = framebuffer_open(module, &fbDev); //打开 Framebuffer 设备
        LOGE_IF(err, "couldn't open framebuffer HAL (%s)", strerror(-err));
        err = gralloc_open(module, &grDev); //打开 gralloc 设备
        LOGE_IF(err, "couldn't open gralloc HAL (%s)", strerror(-err));
        // bail out if we can't initialize the modules
        if (!fbDev || !grDev)
            return;
        mUpdateOnDemand = (fbDev->setUpdateRect != 0);
        //初始化 buffer FIFO
        mNumBuffers = 2;
        mNumFreeBuffers = 2;
        mBufferHead = mNumBuffers-1;
        //初始化连接缓冲区
 buffers[0] = new NativeBuffer(
                fbDev->width, fbDev->height, fbDev->format, GRALLOC_USAGE_HW_FB);
        buffers[1] = new NativeBuffer(
                fbDev->width, fbDev->height, fbDev->format, GRALLOC_USAGE_HW_FB);
        err = grDev->alloc(grDev,
                fbDev->width, fbDev->height, fbDev->format,
                GRALLOC_USAGE_HW_FB, &buffers[0]->handle, &buffers[0]->stride);
        LOGE_IF(err, "fb buffer 0 allocation failed w=%d, h=%d, err=%s",
                fbDev->width, fbDev->height, strerror(-err));
        //从 gralloc 设备中分配内存
        err = grDev->alloc(grDev,
                fbDev->width, fbDev->height, fbDev->format,
                GRALLOC_USAGE_HW_FB, &buffers[1]->handle, &buffers[1]->stride);
        LOGE_IF(err, "fb buffer 1 allocation failed w=%d, h=%d, err=%s",
                fbDev->width, fbDev->height, strerror(-err));
        //从 Framebuffer 设备中获得常量
        const_cast<uint32_t&>(android_native_window_t::flags) = fbDev->flags;
        const_cast<float&>(android_native_window_t::xdpi) = fbDev->xdpi;
        const_cast<float&>(android_native_window_t::ydpi) = fbDev->ydpi;
        const_cast<int&>(android_native_window_t::minSwapInterval) =
            fbDev->minSwapInterval;
```

```
            const_cast<int&>(android_native_window_t::maxSwapInterval) =
                fbDev->maxSwapInterval;
    } else {
        LOGE("Couldn't get gralloc module");
    }
    //赋值各个处理函数的指针
    android_native_window_t::setSwapInterval = setSwapInterval;
    android_native_window_t::dequeueBuffer = dequeueBuffer;
    android_native_window_t::lockBuffer = lockBuffer;
    android_native_window_t::queueBuffer = queueBuffer;
    android_native_window_t::query = query;
    android_native_window_t::perform = perform;
}
```

在函数 FramebufferNativeWindow()中使用了双显示区缓冲方式，具体初始化过程如图 18-2 所示。

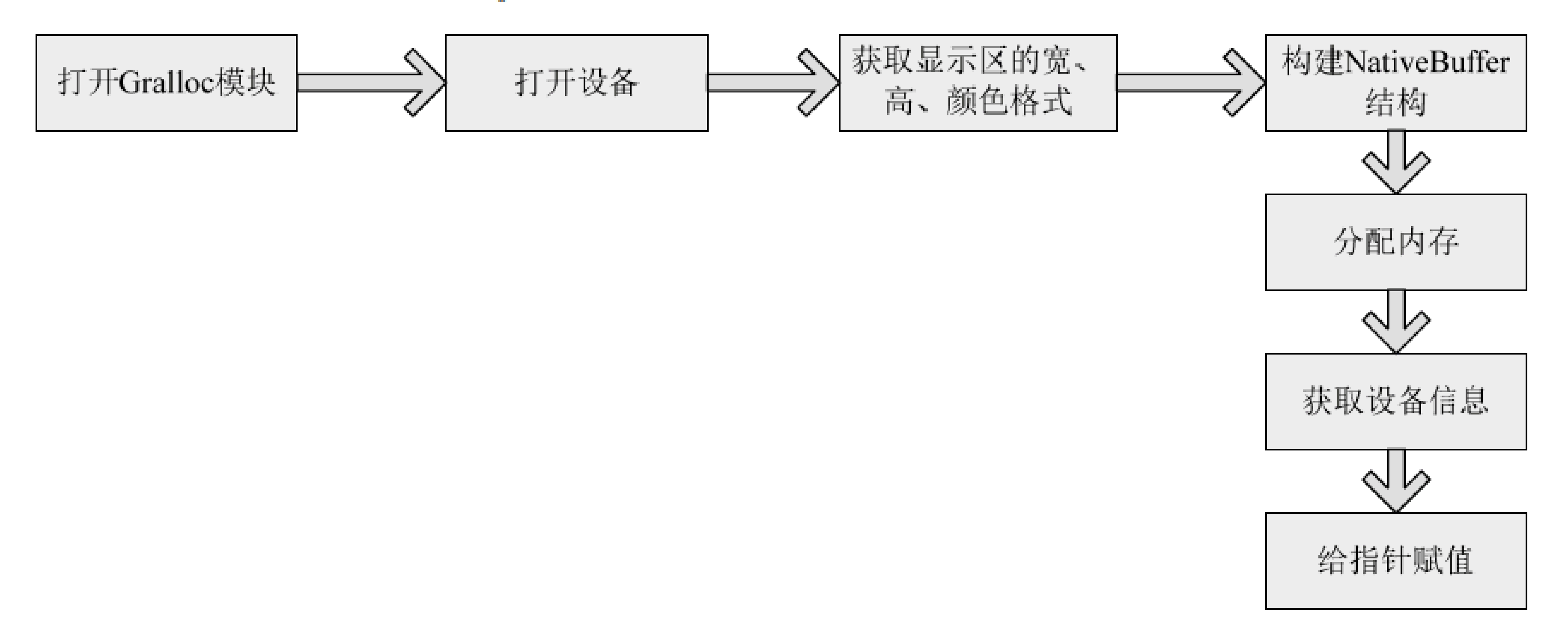


图 18-2　初始化流程图

图 18-2 所示流程的具体描述如下。

☑　打开 Gralloc 模块，然后打开设备 framebuffer_device_t 和 alloc_device_t。
☑　从设备 framebuffer_device_t 中获取显示区的宽、高、颜色格式，构建 NativeBuffer 结构。
☑　从设备 alloc_device_t 中分配内存到 NativeBuffer 句柄。
☑　获取 framebuffer_device_t 设备中的其他信息。
☑　分别给指针 dequeueBuffer、lockBuffer、queueBuffer、query 和 perform 赋值。

18.3.3　显示缓冲区的分配

在文件 frameworks/base/libs/ui/GraphicBufferAllocator.cpp 中，通过调用 Gralloc 模块和 gralloc_ module_t 模块显示缓冲区的分配，此文件的核心代码如下所示。

```
status_t GraphicBufferAllocator::alloc(uint32_t w, uint32_t h, PixelFormat format,
            int usage, buffer_handle_t* handle, int32_t* stride)
{
    // make sure to not allocate a 0 x 0 buffer
    w = clamp(w);
    h = clamp(h);

    // we have a h/w allocator and h/w buffer is requested
    status_t err;
```

```
    if (usage & GRALLOC_USAGE_HW_MASK) {
        err = mAllocDev->alloc(mAllocDev, w, h, format, usage, handle, stride);
    } else {
        err = sw_gralloc_handle_t::alloc(w, h, format, usage, handle, stride);
    }

    LOGW_IF(err, "alloc(%u, %u, %d, %08x, ...) failed %d (%s)",
            w, h, format, usage, err, strerror(-err));

    if (err == NO_ERROR) {
        Mutex::Autolock _l(sLock);
        KeyedVector<buffer_handle_t, alloc_rec_t>& list(sAllocList);
        alloc_rec_t rec;
        rec.w = w;
        rec.h = h;
        rec.format = format;
        rec.usage = usage;
        rec.vaddr = 0;
        rec.size = h * stride[0] * bytesPerPixel(format);
        list.add(*handle, rec);
    } else {
        String8 s;
        dump(s);
        LOGD("%s", s.string());
    }

    return err;
}
```

其中 mAllocDev 是一个 alloc_device_t 设备类型，当调用者具有 GRALLOC_USAGE_HW_MASK 标志时，alloc_device_t 会调用分配一个内存，否则将从软件中分配一个内存。

18.3.4　显示缓冲映射

在文件 frameworks/base/libs/ui/GraphicBufferMapper.cpp 中通过调用 Gralloc 模块的方式显示缓冲的映射，并在里面注册了显示的缓冲内容，使用完毕后可以注销显示的缓冲内容。文件 GraphicBufferMapper.cpp 的核心代码如下所示。

```
//注册显示的缓冲内容
status_t GraphicBufferMapper::registerBuffer(buffer_handle_t handle)
{
    status_t err;
    if (sw_gralloc_handle_t::validate(handle) < 0) {
        err = mAllocMod->registerBuffer(mAllocMod, handle);
    } else {
        err = sw_gralloc_handle_t::registerBuffer((sw_gralloc_handle_t*)handle);
    }
    LOGW_IF(err, "registerBuffer(%p) failed %d (%s)",
            handle, err, strerror(-err));
    return err;
}
```

```
//注销显示的缓冲内容
status_t GraphicBufferMapper::unregisterBuffer(buffer_handle_t handle)
{
    status_t err;
    if (sw_gralloc_handle_t::validate(handle) < 0) {
        err = mAllocMod->unregisterBuffer(mAllocMod, handle);
    } else {
        err = sw_gralloc_handle_t::unregisterBuffer((sw_gralloc_handle_t*)handle);
    }
    LOGW_IF(err, "unregisterBuffer(%p) failed %d (%s)",
            handle, err, strerror(-err));
    return err;
}
//锁定
status_t GraphicBufferMapper::lock(buffer_handle_t handle,
        int usage, const Rect& bounds, void** vaddr)
{
    status_t err;
    if (sw_gralloc_handle_t::validate(handle) < 0) {
        err = mAllocMod->lock(mAllocMod, handle, usage,
                bounds.left, bounds.top, bounds.width(), bounds.height(),
                vaddr);
    } else {
        err = sw_gralloc_handle_t::lock((sw_gralloc_handle_t*)handle, usage,
                bounds.left, bounds.top, bounds.width(), bounds.height(),
                vaddr);
    }
    LOGW_IF(err, "lock(...) failed %d (%s)", err, strerror(-err));
    return err;
}
//解锁
status_t GraphicBufferMapper::unlock(buffer_handle_t handle)
{
    status_t err;
    if (sw_gralloc_handle_t::validate(handle) < 0) {
        err = mAllocMod->unlock(mAllocMod, handle);
    } else {
        err = sw_gralloc_handle_t::unlock((sw_gralloc_handle_t*)handle);
    }
    LOGW_IF(err, "unlock(...) failed %d (%s)", err, strerror(-err));
    return err;
}
```

其中 mAllocDev 是一个 alloc_device_t 设备类型，根据句柄的范围可以从 Gralloc 模块中注册 Buffer，也可以从软件中注册 Buffer。

18.3.5 分析管理库文件 LayerBuffer.cpp

管理库 SurfaceFlinger 中也调用了 Gralloc 模块，调用路径为 frameworks/base/libs/surfaceflinger/LayerBuffer.cpp。SurfaceFlinger 按英文翻译过来就是 Surface 投递者。SurfaceFlinger 的构成并不复杂，复杂的是它的客户



端建构。SurfaceFlinger 的主要功能如下。

☑ 将 Layers（Surfaces）内容刷新到屏幕上。
☑ 维持 Layer 的 Z-order 序列，并对 Layer 最终输出做出裁剪计算。
☑ 响应 Client 要求，创建 Layer 与客户端的 Surface 建立连接。
☑ 接收 Client 要求，修改输出大小、Alpha 等 Layer 属性。

SurfaceFlinger 的基本组成框架如图 18-3 所示。

SurfaceFlinger 的管理对象如下。

☑ mClientsMap：管理客户端与服务端的连接。
☑ ISurface、IsurfaceComposer：AIDL 调用接口实例。
☑ mLayerMap：服务端的 Surface 的管理对象。
☑ mCurrentState.layersSortedByZ：以 Surface 的 Z-order 序列排列的 Layer 数组。
☑ graphicPlane：缓冲区输出管理。
☑ OpenGL ES：图形计算、图像合成等图形库。

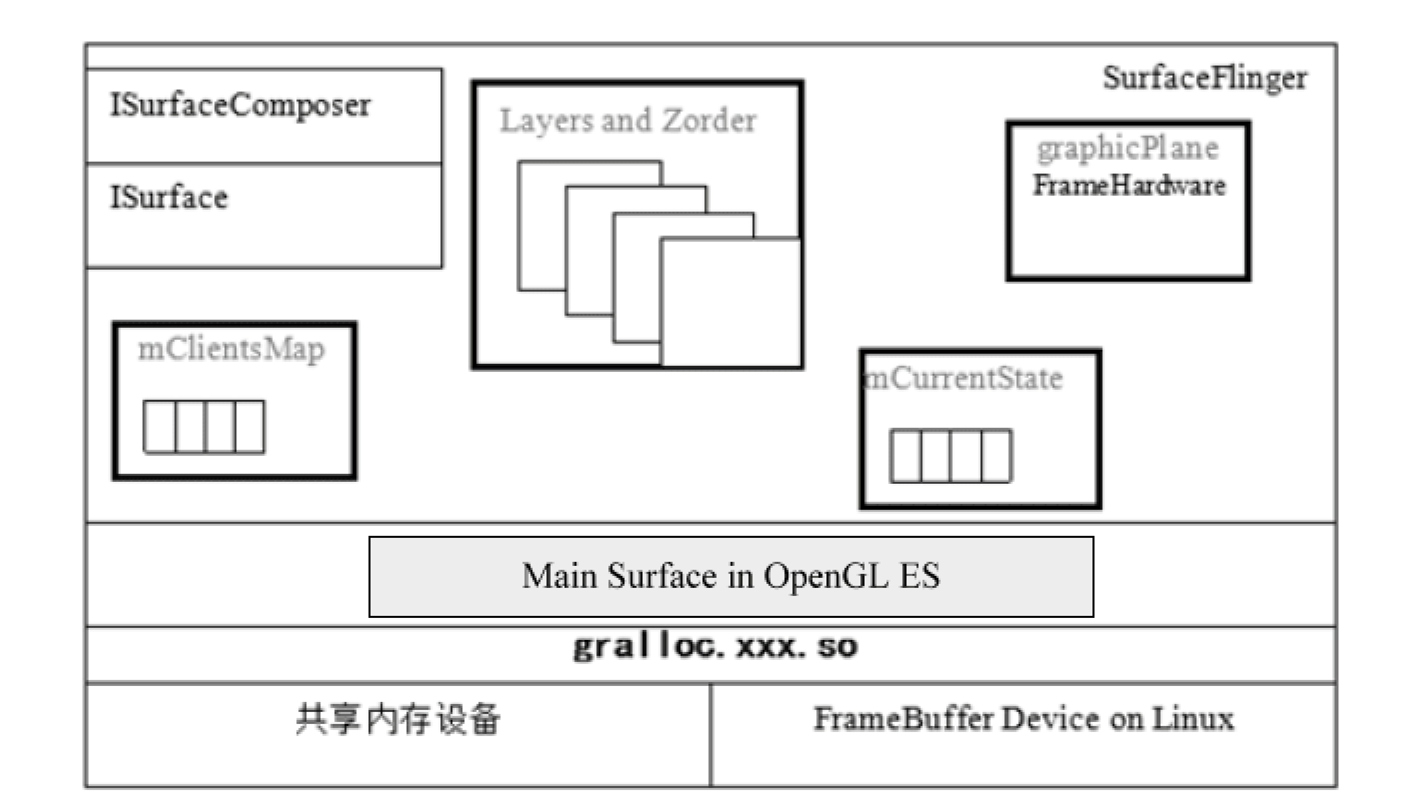


图 18-3　SurfaceFlinger 的基本组成框架

☑ gralloc.xxx.so：是一个和平台相关的图形缓冲区管理器。
☑ pmem Device：提供共享内存，在这里只是在 gralloc.xxx.so 可见，在上层被 gralloc.xxx.so 抽象了。

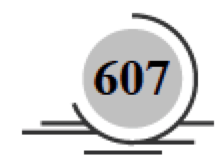
在文件 LayerBuffer.cpp 中定义一个 Buffer 类，并为其定义了构造函数，构造函数的实现代码如下所示。

```
LayerBuffer::Buffer::Buffer(const ISurface::BufferHeap& buffers, ssize_t offset)
    : mBufferHeap(buffers)
{
    NativeBuffer& src(mNativeBuffer);
    src.crop.l = 0;
    src.crop.t = 0;
    src.crop.r = buffers.w;
    src.crop.b = buffers.h;
    src.img.w = buffers.hor_stride ?: buffers.w;
    src.img.h = buffers.ver_stride ?: buffers.h;
    src.img.format = buffers.format;
    src.img.offset = offset;
    src.img.base = buffers.heap->base();
    src.img.fd = buffers.heap->heapID();
}
```

在上述代码中，调用了 gralloc_module_t 的可选实现的函数指针 perform，如果在当前使用的 Gralloc 模块中实现了这个函数指针时则在此调用函数。

18.4　Goldfish 中的 FrameBuffer 驱动程序详解

在 Android 模拟器中，使用的驱动程序是 Goldfish 和 FrameBuffer 驱动程序，使用的硬件抽象层是 Gralloc



模块。Gralloc 模块既可以被模拟器使用，也可以给实际硬件系统使用。Goldfish 中的 FrameBuffer 驱动程序保存在文件 drivers/video/goldfishfb.c 中，此文件的主要实现代码如下所示。

```
//验证函数
static int goldfish_fb_check_var(struct fb_var_screeninfo *var, struct fb_info *info)
{
        if((var->rotate & 1) != (info->var.rotate & 1)) {
                if((var->xres != info->var.yres) ||
                     (var->yres != info->var.xres) ||
                     (var->xres_virtual != info->var.yres) ||
                     (var->yres_virtual > info->var.xres * 2) ||
                     (var->yres_virtual < info->var.xres )) {
                        return -EINVAL;
                }
        }
        else {
                if((var->xres != info->var.xres) ||
                     (var->yres != info->var.yres) ||
                     (var->xres_virtual != info->var.xres) ||
                     (var->yres_virtual > info->var.yres * 2) ||
                     (var->yres_virtual < info->var.yres )) {
                        return -EINVAL;
                }
        }
        if((var->xoffset != info->var.xoffset) ||
             (var->bits_per_pixel != info->var.bits_per_pixel) ||
             (var->grayscale != info->var.grayscale)) {
                return -EINVAL;
        }
        return 0;
}
//驱动程序的初始化函数
static int goldfish_fb_probe(struct platform_device *pdev)
{
        int ret;
        struct resource *r;
        struct goldfish_fb *fb;
        size_t framesize;
        uint32_t width, height;
        dma_addr_t fbpaddr;

        fb = kzalloc(sizeof(*fb), GFP_KERNEL);
        if(fb == NULL) {
                ret = -ENOMEM;
                goto err_fb_alloc_failed;
        }
        spin_lock_init(&fb->lock);
        init_waitqueue_head(&fb->wait);
        platform_set_drvdata(pdev, fb);

        r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
```

```
    if(r == NULL) {
        ret = -ENODEV;
        goto err_no_io_base;
    }
    fb->reg_base = IO_ADDRESS(r->start - IO_START);

    fb->irq = platform_get_irq(pdev, 0);
    if(fb->irq < 0) {
        ret = -ENODEV;
        goto err_no_irq;
    }

    width = readl(fb->reg_base + FB_GET_WIDTH);
    height = readl(fb->reg_base + FB_GET_HEIGHT);

    fb->fb.fbops            = &goldfish_fb_ops;
    fb->fb.flags            = FBINFO_FLAG_DEFAULT;
    fb->fb.pseudo_palette       = fb->cmap;
    //strncpy(fb->fb.fix.id, clcd_name, sizeof(fb->fb.fix.id));
    fb->fb.fix.type         = FB_TYPE_PACKED_PIXELS;
    fb->fb.fix.visual = FB_VISUAL_TRUECOLOR;
    fb->fb.fix.line_length = width * 2; //RGB565 每个像素占用 16 位，两个字节
    fb->fb.fix.accel  = FB_ACCEL_NONE;
    fb->fb.fix.ypanstep = 1;

    fb->fb.var.xres         = width;//实际显示区域
    fb->fb.var.yres         = height;
    fb->fb.var.xres_virtual     = width; //虚拟显示区域
    fb->fb.var.yres_virtual     = height * 2;
    fb->fb.var.bits_per_pixel = 16;
    fb->fb.var.activate     = FB_ACTIVATE_NOW;
    fb->fb.var.height       = readl(fb->reg_base + FB_GET_PHYS_HEIGHT);
    fb->fb.var.width= readl(fb->reg_base + FB_GET_PHYS_WIDTH);

    fb->fb.var.red.offset = 11;
    fb->fb.var.red.length = 5;
    fb->fb.var.green.offset = 5;
    fb->fb.var.green.length = 6;
    fb->fb.var.blue.offset = 0;
    fb->fb.var.blue.length = 5;
  //显示缓冲区大小
    framesize = width * height * 2 * 2;
  //进行内存映射
    fb->fb.screen_base = dma_alloc_writecombine(&pdev->dev, framesize,
                                        &fbpaddr, GFP_KERNEL);
    printk("allocating frame buffer %d * %d, got %p\n", width, height, fb->fb.screen_base);
    if(fb->fb.screen_base == 0) {
        ret = -ENOMEM;
        goto err_alloc_screen_base_failed;
    }
    fb->fb.fix.smem_start = fbpaddr;
```

```
	fb->fb.fix.smem_len = framesize;

	ret = fb_set_var(&fb->fb, &fb->fb.var);
	if(ret)
		goto err_fb_set_var_failed;

	ret = request_irq(fb->irq, goldfish_fb_interrupt, IRQF_SHARED, pdev->name, fb);
	if(ret)
		goto err_request_irq_failed;

	writel(FB_INT_BASE_UPDATE_DONE, fb->reg_base + FB_INT_ENABLE);
	goldfish_fb_pan_display(&fb->fb.var, &fb->fb); // updates base

	ret = register_framebuffer(&fb->fb);
	if(ret)
		goto err_register_framebuffer_failed;

#ifdef CONFIG_ANDROID_POWER
	fb->early_suspend.suspend = goldfish_fb_early_suspend;
	fb->early_suspend.resume = goldfish_fb_late_resume;
	android_register_early_suspend(&fb->early_suspend);
#endif

	return 0;

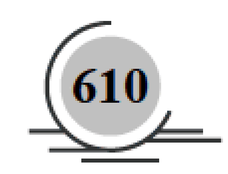
//错误处理
err_register_framebuffer_failed:
	free_irq(fb->irq, fb);
err_request_irq_failed:
err_fb_set_var_failed:
	dma_free_writecombine(&pdev->dev, framesize, fb->fb.screen_base, fb->fb.fix.smem_start);
err_alloc_screen_base_failed:
err_no_irq:
err_no_io_base:
	kfree(fb);
err_fb_alloc_failed:
	return ret;
}
```

上述代码通过 FrameBuffer 驱动程序实现了对 RGB565 颜色空间的支持，其中虚拟显示的 y 值是实际显示的两倍，这样就实现了双缓存功能。

18.5 使用 Gralloc 模块的驱动程序

在 Android 系统中，不同的硬件有不同的硬件图形加速设备和缓冲内存实现方法。Android Gralloc 动态库抽象的任务是消除不同设备之间的差别，在上层看来都是同样的方法和对象。在 Module 层隐藏缓冲区操作细节。Android 使用了动态链接库 gralloc.xxx.so 来封装底层细节。

默认 Gralloc 模块的实现源码保存在 hardware/libhardware/modules/gralloc/目录下，Android Gralloc 主要



有如下 3 个实现文件。

☑ gralloc.cpp：实现了 gralloc_module_t 模块和 alloc_device_t 设备。
☑ mapper.cpp：实现了工具函数。
☑ framebuffer.cpp：实现了 alloc_device_t 设备。

下面将详细讲解上述 3 个文件的具体实现过程。

18.5.1　文件 gralloc.cpp

（1）定义函数 gralloc_device_open()，这是一个模块打开函数，具体实现代码如下所示。

```
int gralloc_device_open(const hw_module_t* module, const char* name,
          hw_device_t** device)
{
    int status = -EINVAL;
    if (!strcmp(name, GRALLOC_HARDWARE_GPU0)) {
        gralloc_context_t *dev;
        dev = (gralloc_context_t*)malloc(sizeof(*dev));

        /* initialize our state here */
        memset(dev, 0, sizeof(*dev));

        /* initialize the procs */
        dev->device.common.tag = HARDWARE_DEVICE_TAG;
        dev->device.common.version = 0;
        dev->device.common.module = const_cast<hw_module_t*>(module);
        dev->device.common.close = gralloc_close;

        dev->device.alloc     = gralloc_alloc;
        dev->device.free      = gralloc_free;

        *device = &dev->device.common;
        status = 0;
    } else {
//打开 framebuffer_device_t 设备
        status = fb_device_open(module, name, device);
    }
    return status;
}
```

（2）定义函数 gralloc_alloc_framebuffer_locked()，对应代码如下所示。

```
static int gralloc_alloc_framebuffer_locked(alloc_device_t* dev,
          size_t size, int usage, buffer_handle_t* pHandle)
{
    private_module_t* m = reinterpret_cast<private_module_t*>(
            dev->common.module);

   // allocate the framebuffer
    if (m->framebuffer == NULL) {
        // initialize the framebuffer, the framebuffer is mapped once
        // and forever.
        int err = mapFrameBufferLocked(m);
```

```
            if (err < 0) {
                return err;
            }
        }
```

函数 gralloc_alloc_framebuffer_locked()的功能是，如果当前没有 Framebuffer-->mapFrameBufferLocked(m)，如果不支持 PAGE_FLIP，则通过软件方法分配 gralloc_alloc_buffer，并且决定使用双 Framebuffer 中的哪个作为缓冲地址。

（3）定义函数 gralloc_alloc_buffer()，主要实现代码如下所示。

```
static int gralloc_alloc_buffer(alloc_device_t* dev,
            size_t size, int usage, buffer_handle_t* pHandle)
{
    int err = 0;
    int flags = 0;

    int fd = -1;
    void* base = 0;
    int offset = 0;
    int lockState = 0;

    size = roundUpToPageSize(size);

#if HAVE_ANDROID_OS // should probably define HAVE_PMEM somewhere

    if (usage & GRALLOC_USAGE_HW_TEXTURE) {
        // enable pmem in that case, so our software GL can fallback to
        // the copybit module.
        flags |= private_handle_t::PRIV_FLAGS_USES_PMEM;
    }

    if (usage & GRALLOC_USAGE_HW_2D) {
        flags |= private_handle_t::PRIV_FLAGS_USES_PMEM;
    }

    if ((flags & private_handle_t::PRIV_FLAGS_USES_PMEM) == 0) {
try_ashmem:
        fd = ashmem_create_region("gralloc-buffer", size);
        if (fd < 0) {
            LOGE("couldn't create ashmem (%s)", strerror(-errno));
            err = -errno;
        }
    } else {
        private_module_t* m = reinterpret_cast<private_module_t*>(
                dev->common.module);

        err = init_pmem_area(m);
        if (err == 0) {
            // PMEM buffers are always mmapped
            base = m->pmem_master_base;
            lockState |= private_handle_t::LOCK_STATE_MAPPED;
```

```
                offset = sAllocator.allocate(size);
                if (offset < 0) {
                    // no more pmem memory
                    err = -ENOMEM;
                } else {
                    struct pmem_region sub = { offset, size };

                   // now create the "sub-heap"
                    fd = open("/dev/pmem", O_RDWR, 0);
                    err = fd < 0 ? fd : 0;

                    // and connect to it
                    if (err == 0)
                        err = ioctl(fd, PMEM_CONNECT, m->pmem_master);

                    // and make it available to the client process
                   if (err == 0)
                        err = ioctl(fd, PMEM_MAP, &sub);

                    if (err < 0) {
                        err = -errno;
                        close(fd);
                        sAllocator.deallocate(offset);
                        fd = -1;
                    }
                    //LOGD_IF(!err, "allocating pmem size=%d, offset=%d", size, offset);
                    memset((char*)base + offset, 0, size);
                }
            } else {
                if ((usage & GRALLOC_USAGE_HW_2D) == 0) {
                    // the caller didn't request PMEM, so we can try something else
                    flags &= ~private_handle_t::PRIV_FLAGS_USES_PMEM;
                    err = 0;
                    goto try_ashmem;
                } else {
                    LOGE("couldn't open pmem (%s)", strerror(-errno));
                }
            }
        }

#else // HAVE_ANDROID_OS

        fd = ashmem_create_region("Buffer", size);
        if (fd < 0) {
            LOGE("couldn't create ashmem (%s)", strerror(-errno));
            err = -errno;
        }

#endif // HAVE_ANDROID_OS

    if (err == 0) {
```

```
            private_handle_t* hnd = new private_handle_t(fd, size, flags);
            hnd->offset = offset;
            hnd->base = int(base)+offset;
            hnd->lockState = lockState;
            *pHandle = hnd;
        }

        LOGE_IF(err, "gralloc failed err=%s", strerror(-err));

        return err;
 }
```

上述代码的实现流程如下。

☑ 如果系统没用 PMEM，则直接赋值 fd = ashmem_create_region("gralloc-buffer", size)。

☑ 如果有 PMEM，则 init_pmem_area(m)。

☑ 获取本次需要的 size：offset = sAllocator.allocate(size)。

☑ 建立 PMEM region：struct pmem_region sub = { offset, size }。

☑ 重新打开：fd = open("/dev/pmem", O_RDWR, 0)。

☑ 链接空间：err = ioctl(fd, PMEM_CONNECT, m->pmem_master)。

☑ 获取句柄：private_handle_t* hnd = new private_handle_t(fd, size, flags)。

在文件 gralloc.cpp 中，结构体类型 private_module_t 扩展了 gralloc_module_t 结构体，此结构体在文件 gralloc_priv.h 中定义，实现代码如下所示。

```
struct private_module_t {
        gralloc_module_t base;

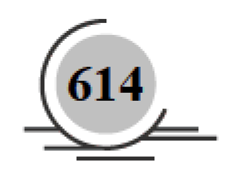
        private_handle_t* framebuffer;
        uint32_t flags;
        uint32_t numBuffers;
        uint32_t bufferMask;
        pthread_mutex_t lock;
        buffer_handle_t currentBuffer;
        int pmem_master;
        void* pmem_master_base;

        struct fb_var_screeninfo info;
        struct fb_fix_screeninfo finfo;
        float xdpi;
        float ydpi;
        float fps;
        enum {
            // flag to indicate we'll post this buffer
            PRIV_USAGE_LOCKED_FOR_POST = 0x80000000
        };
```

18.5.2 文件 mapper.cpp

在文件 mapper.cpp 中定义的函数是结构体 gralloc_module_t 的具体实现，此文件的具体实现流程如下。

（1）定义函数 gralloc_register_buffer()，功能是建立一个新的 private_handle_t 对象，如果不是本进程对



象则赋初值，其实现代码如下所示。

```
int gralloc_register_buffer(gralloc_module_t const* module,
        buffer_handle_t handle)
{
    if (private_handle_t::validate(handle) < 0)
        return -EINVAL;
    private_handle_t* hnd = (private_handle_t*)handle;
    if (hnd->pid != getpid()) {
        hnd->base = 0;
        hnd->lockState   = 0;
        hnd->writeOwner = 0;
    }
    return 0;
}
```

（2）定义函数 gralloc_unregister_buffer()，通过 validate 判断 handle 是否合法。具体实现代码如下所示。

```
int gralloc_unregister_buffer(gralloc_module_t const* module,
        buffer_handle_t handle)
{
    if (private_handle_t::validate(handle) < 0)
        return -EINVAL;
    private_handle_t* hnd = (private_handle_t*)handle;

    LOGE_IF(hnd->lockState & private_handle_t::LOCK_STATE_READ_MASK,
            "[unregister] handle %p still locked (state=%08x)",
            hnd, hnd->lockState);

    // never unmap buffers that were created in this process
    if (hnd->pid != getpid()) {
        if (hnd->lockState & private_handle_t::LOCK_STATE_MAPPED) {
            gralloc_unmap(module, handle);
        }
        hnd->base = 0;
        hnd->lockState   = 0;
        hnd->writeOwner = 0;
    }
    return 0;
}
```

18.5.3　文件 framebuffer.cpp

文件 framebuffer.cpp 用于实现设备 framebuffer_device_t，其核心代码和 Donut 之前版本的 EGLDisplay Surface.cpp 文件的实现类似，不同的是在文件 framebuffer.cpp 中使用双缓冲的实现方式。下面将详细讲解此文件的具体实现流程。

（1）定义函数 fb_device_open()用于初始化设备 framebuffer_device_t，实现代码如下所示。

```
int fb_device_open(hw_module_t const* module, const char* name,
        hw_device_t** device)
{
    int status = -EINVAL;
    if (!strcmp(name, GRALLOC_HARDWARE_FB0)) {
```

```
        alloc_device_t* gralloc_device;
        status = gralloc_open(module, &gralloc_device);
        if (status < 0)
            return status;

        /* initialize our state here */
        fb_context_t *dev = (fb_context_t*)malloc(sizeof(*dev));
        memset(dev, 0, sizeof(*dev));

        /* initialize the procs */
        dev->device.common.tag = HARDWARE_DEVICE_TAG;
        dev->device.common.version = 0;
        dev->device.common.module = const_cast<hw_module_t*>(module);
        dev->device.common.close = fb_close;
        dev->device.setSwapInterval = fb_setSwapInterval;
        dev->device.post = fb_post;
        dev->device.setUpdateRect = 0;
           private_module_t* m = (private_module_t*)module;
        status = mapFrameBuffer(m);//映射 FrameBuffer 设备
        if (status >= 0) {//填充设备 framebuffer_device_t 的各个内容
            int stride = m->finfo.line_length / (m->info.bits_per_pixel >> 3);
            const_cast<uint32_t&>(dev->device.flags) = 0;
            const_cast<uint32_t&>(dev->device.width) = m->info.xres;
            const_cast<uint32_t&>(dev->device.height) = m->info.yres;
            const_cast<int&>(dev->device.stride) = stride;
            const_cast<int&>(dev->device.format) = HAL_PIXEL_FORMAT_RGB_565;
            const_cast<float&>(dev->device.xdpi) = m->xdpi;
            const_cast<float&>(dev->device.ydpi) = m->ydpi;
            const_cast<float&>(dev->device.fps) = m->fps;
            const_cast<int&>(dev->device.minSwapInterval) = 1;
            const_cast<int&>(dev->device.maxSwapInterval) = 1;
            *device = &dev->device.common;
        }
    }
    return status;
}
```

（2）定义函数 mapFrameBufferLocked()，功能是实现 FrameBuffer 设备的真正功能，具体实现代码如下所示。

```
int mapFrameBufferLocked(struct private_module_t* module)
{
    // already initialized...
    if (module->framebuffer) {
        return 0;
    }

    char const * const device_template[] = {
            "/dev/graphics/fb%u",
            "/dev/fb%u",
            0 };
```

```
int fd = -1;
int i=0;
char name[64];

while ((fd==-1) && device_template[i]) {
      snprintf(name, 64, device_template[i], 0);
      fd = open(name, O_RDWR, 0);
      i++;
}
if (fd < 0)
      return -errno;

struct fb_fix_screeninfo finfo;
if (ioctl(fd, FBIOGET_FSCREENINFO, &finfo) == -1)
      return -errno;

struct fb_var_screeninfo info;
if (ioctl(fd, FBIOGET_VSCREENINFO, &info) == -1)
      return -errno;

info.reserved[0] = 0;
info.reserved[1] = 0;
info.reserved[2] = 0;
info.xoffset = 0;
info.yoffset = 0;
info.activate = FB_ACTIVATE_NOW;
info.bits_per_pixel = 16;
info.red.offset       = 11;
info.red.length       = 5;
info.green.offset     = 5;
info.green.length     = 6;
info.blue.offset      = 0;
info.blue.length      = 5;
info.transp.offset    = 0;
info.transp.length    = 0;
info.yres_virtual = info.yres * NUM_BUFFERS;
uint32_t flags = PAGE_FLIP;
if (ioctl(fd, FBIOPUT_VSCREENINFO, &info) == -1) {
      info.yres_virtual = info.yres;
      flags &= ~PAGE_FLIP;
      LOGW("FBIOPUT_VSCREENINFO failed, page flipping not supported");
}

if (info.yres_virtual < info.yres * 2) {
      // we need at least 2 for page-flipping
      info.yres_virtual = info.yres;
      flags &= ~PAGE_FLIP;
      LOGW("page flipping not supported (yres_virtual=%d, requested=%d)",
              info.yres_virtual, info.yres*2);
}
```

```
if (ioctl(fd, FBIOGET_VSCREENINFO, &info) == -1)
    return -errno;

int refreshRate = 1000000000000000LLU /
(
        uint64_t( info.upper_margin + info.lower_margin + info.yres )
        * ( info.left_margin  + info.right_margin + info.xres )
        * info.pixclock
);

if (refreshRate == 0) {
    // bleagh, bad info from the driver
    refreshRate = 60*1000;  // 60 Hz
}

if (int(info.width) <= 0 || int(info.height) <= 0) {
    // the driver doesn't return that information
    // default to 160 dpi
    info.width  = ((info.xres * 25.4f)/160.0f + 0.5f);
    info.height = ((info.yres * 25.4f)/160.0f + 0.5f);
}

float xdpi = (info.xres * 25.4f) / info.width;
float ydpi = (info.yres * 25.4f) / info.height;
float fps  = refreshRate / 1000.0f;

LOGI(   "using (fd=%d)\n"
        "id           = %s\n"
        "xres         = %d px\n"
        "yres         = %d px\n"
        "xres_virtual = %d px\n"
        "yres_virtual = %d px\n"
        "bpp          = %d\n"
        "r            = %2u:%u\n"
        "g            = %2u:%u\n"
        "b            = %2u:%u\n",
        fd,
        finfo.id,
        info.xres,
        info.yres,
        info.xres_virtual,
        info.yres_virtual,
        info.bits_per_pixel,
        info.red.offset, info.red.length,
        info.green.offset, info.green.length,
        info.blue.offset, info.blue.length
);

LOGI(   "width        = %d mm (%f dpi)\n"
        "height       = %d mm (%f dpi)\n"
        "refresh rate = %.2f Hz\n",
```

```
                info.width,    xdpi,
                info.height, ydpi,
                fps
        );
        if (ioctl(fd, FBIOGET_FSCREENINFO, &finfo) == -1)
            return -errno;
        if (finfo.smem_len <= 0)
            return -errno;
        module->flags = flags;
        module->info = info;
        module->finfo = finfo;
        module->xdpi = xdpi;
        module->ydpi = ydpi;
        module->fps = fps;

        int err;
        size_t fbSize = roundUpToPageSize(finfo.line_length * info.yres_virtual);
        module->framebuffer = new private_handle_t(dup(fd), fbSize,
                private_handle_t::PRIV_FLAGS_USES_PMEM);
        module->numBuffers = info.yres_virtual / info.yres;
        module->bufferMask = 0;
        void* vaddr = mmap(0, fbSize, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        if (vaddr == MAP_FAILED) {
            LOGE("Error mapping the framebuffer (%s)", strerror(errno));
            return -errno;
        }
        module->framebuffer->base = intptr_t(vaddr);
        memset(vaddr, 0, fbSize);
        return 0;
    }
```

上述代码需要完成如下工作。

- ☑ 打开 framebuffer 设备。
- ☑ 判断是否支持 PAGE_FLIP。
- ☑ 计算刷新率。
- ☑ 打印 gralloc 信息。
- ☑ 填充 private_module_t。

（3）定义函数 fb_post()，功能是将某个缓冲区显示在屏幕上，具体实现代码如下所示。

```
static int fb_post(struct framebuffer_device_t* dev, buffer_handle_t buffer)
  {
        if (private_handle_t::validate(buffer) < 0)
            return -EINVAL;
        fb_context_t* ctx = (fb_context_t*)dev;
        private_handle_t const* hnd = reinterpret_cast<private_handle_t const*>(buffer);
        private_module_t* m = reinterpret_cast<private_module_t*>(
                dev->common.module);

        if (m->currentBuffer) {
            m->base.unlock(&m->base, m->currentBuffer);
           m->currentBuffer = 0;
        }
```

```
        if (hnd->flags & private_handle_t::PRIV_FLAGS_FRAMEBUFFER) {
            m->base.lock(&m->base, buffer,
                    private_module_t::PRIV_USAGE_LOCKED_FOR_POST,
                    0, 0, m->info.xres, m->info.yres, NULL);

            const size_t offset = hnd->base - m->framebuffer->base;
            m->info.activate = FB_ACTIVATE_VBL;
            m->info.yoffset = offset / m->finfo.line_length;
            if (ioctl(m->framebuffer->fd, FBIOPUT_VSCREENINFO, &m->info) == -1) {
                LOGE("FBIOPUT_VSCREENINFO failed");
                m->base.unlock(&m->base, buffer);
                return -errno;
            }
            m->currentBuffer = buffer;
        } else {
            // If we can't do the page_flip, just copy the buffer to the front
            // FIXME: use copybit HAL instead of memcpy

            void* fb_vaddr;
            void* buffer_vaddr;

            m->base.lock(&m->base, m->framebuffer,
                    GRALLOC_USAGE_SW_WRITE_RARELY,
                    0, 0, m->info.xres, m->info.yres,
                    &fb_vaddr);
            m->base.lock(&m->base, buffer,
                    GRALLOC_USAGE_SW_READ_RARELY,
                    0, 0, m->info.xres, m->info.yres,
                        &buffer_vaddr);
            memcpy(fb_vaddr, buffer_vaddr, m->finfo.line_length * m->info.yres);

            m->base.unlock(&m->base, buffer);
            m->base.unlock(&m->base, m->framebuffer);
        }
        return 0;
    }
```

在上述代码中，会检查 buffer 是否合法，并进行相应的类型转换处理。如果 currentbuffer 非空则 unlock。

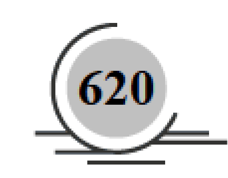
注意： 上述代码非常科学，屏幕上显示的内容其实是通过硬件 DMA 读取显示缓冲区的数据，而在程序中需要写入显示缓冲区的数据。为了避免上述两个步骤同时进行，Gralloc 使用了如下科学合理的处理方式。

☑ 锁定其中的一个后再写内容，写完后解锁。

☑ 在解锁期间，此显示缓冲区不能被硬件 DMA 获取，在期间另一个缓冲区被解锁，此时可以显示到屏幕上。

18.6 MSM 高通处理器中的显示驱动

MSM 处理器的入口源文件是 drivers/staging/msm/msm_fb.c，这是一个标准的 FrameBuffer 驱动程序，此驱动使用了 RGB565 颜色空间，使用 2 倍和实际显示区内存作为虚拟显示区。下面将详细介绍文件 msm_fb.c



的实现流程。

18.6.1　msm fb 设备的文件操作函数接口

定义接口 fb_ops msm_fb_ops，这是高通 msm fb 设备的文件操作函数接口，具体代码如下所示。

```
static struct fb_ops msm_fb_ops = {
.owner = THIS_MODULE,
.fb_open = msm_fb_open,
.fb_release = msm_fb_release,
.fb_read = NULL,
.fb_write = NULL,
.fb_cursor = NULL,
.fb_check_var = msm_fb_check_var,              /*  参数检查  */
.fb_set_par = msm_fb_set_par,                  /*  设置显示相关参数  */
.fb_setcolreg = NULL,                          /* set color register */
.fb_blank = NULL,                              /* blank display */
.fb_pan_display = msm_fb_pan_display,          /*  显示  */
.fb_fillrect = msm_fb_fillrect,                /* Draws a rectangle */
.fb_copyarea = msm_fb_copyarea,                /* Copy data from area to another */
.fb_imageblit = msm_fb_imageblit,              /* Draws a image to the display */
.fb_cursor = NULL,
.fb_rotate = NULL,
.fb_sync = NULL,                               /* wait for blit idle, optional */
.fb_ioctl = msm_fb_ioctl,                      /* perform fb specific ioctl (optional) */
.fb_mmap = NULL,
};
```

18.6.2　高通 msm fb 的 driver 接口

定义接口 platform_driver msm_fb_driver，这是高通 msm fb 的 driver 接口，具体代码如下所示。

```
static struct platform_driver msm_fb_driver = {
.probe = msm_fb_probe,                         //驱动探测函数
.remove = msm_fb_remove,
#ifndef CONFIG_ANDROID_POWER
.suspend = msm_fb_suspend,
.suspend_late = NULL,
.resume_early = NULL,
.resume = msm_fb_resume,
#endif
.shutdown = NULL,
.driver = {
		/* Driver name must match the device name added in platform.c. */
		.name = "msm_fb",
		},
};
```

18.6.3　特殊的 ioctl

和标准的 FrameBuffer 驱动程序相比，MSM 在驱动中增加了特殊的 ioctl，对应代码如下所示。

```
void msm_fb_add_device(struct platform_device *pdev)
{
	struct msm_fb_panel_data *pdata;
	struct platform_device *this_dev = NULL;
	struct fb_info *fbi;
	struct msm_fb_data_type *mfd = NULL;
	u32 type, id, fb_num;

	if (!pdev)
		return;
	id = pdev->id;
	pdata = pdev->dev.platform_data;
	if (!pdata)
		return;
	type = pdata->panel_info.type;
	fb_num = pdata->panel_info.fb_num;

	if (fb_num <= 0)
		return;

	if (fbi_list_index >= MAX_FBI_LIST) {
		printk(KERN_ERR "msm_fb: no more framebuffer info list!\n");
		return;
	}
	/**
	 * alloc panel device data
	 */
	this_dev = msm_fb_device_alloc(pdata, type, id);

	if (!this_dev) {
		printk(KERN_ERR
		"%s: msm_fb_device_alloc failed!\n", __func__);
		return;
	}

	/**
	 * alloc framebuffer info + par data
	 */
	fbi = framebuffer_alloc(sizeof(struct msm_fb_data_type), NULL);
	if (fbi == NULL) {
		platform_device_put(this_dev);
		printk(KERN_ERR "msm_fb: can't alloca framebuffer info data!\n");
		return;
	}

	mfd = (struct msm_fb_data_type *)fbi->par;
	mfd->key = MFD_KEY;
	mfd->fbi = fbi;
	mfd->panel.type = type;
	mfd->panel.id = id;
	mfd->fb_page = fb_num;
```

```
    mfd->index = fbi_list_index;
    mfd->mdp_fb_page_protection = MDP_FB_PAGE_PROTECTION_WRITECOMBINE;
    mfd->pdev = this_dev;
    mfd_list[mfd_list_index++] = mfd;
    fbi_list[fbi_list_index++] = fbi;
    platform_set_drvdata(this_dev, mfd);

    if (platform_device_add(this_dev)) {
        printk(KERN_ERR "msm_fb: platform_device_add failed!\n");
        platform_device_put(this_dev);
        framebuffer_release(fbi);
        fbi_list_index--;
        return;
    }
}
```

18.7　MSM 中的 Gralloc 驱动程序详解

在 MSM 处理器中，重新实现了 Gralloc 模块的架构，此 Gralloc 模块是基于 FrameBuffer 和 Pmem 驱动实现的。MSM 平台中和 Gralloc 模块相关的文件目录如下。

☑　hardware/msm7k/libgralloc：MSM7 系列的实现文件。

☑　hardware/msm7k/libgralloc-qsd8k：QSD8K 系列的实现文件。

在 MSM 中实现 Gralloc 模块的方法和在 Android 系统中的实现方法类似，主要变化是使用了 Pmem 部分。

18.7.1　文件 gralloc.cpp

在文件 gralloc.cpp 中，使用 Gralloc 中的结构体 private_module_t 来扩展里面的结构体 private_ module_t。结构体 private_module_t 的实现文件是 gralloc_priv.h，在里面包含了和上下文有关的信息。下面将简单分析文件 gralloc.cpp 的实现流程。

在文件 gralloc.cpp 中首先需要定义结构体 HAL_MODULE_INFO_SYM，具体代码如下所示。

```
struct private_module_t HAL_MODULE_INFO_SYM = {
    base: {
        common: {
            tag: HARDWARE_MODULE_TAG,
            version_major: 1,
            version_minor: 0,
            id: GRALLOC_HARDWARE_MODULE_ID,
            name: "Graphics Memory Allocator Module",
            author: "The Android Open Source Project",
            methods: &gralloc_module_methods
        },
        registerBuffer: gralloc_register_buffer,
        unregisterBuffer: gralloc_unregister_buffer,
        lock: gralloc_lock,
        unlock: gralloc_unlock,
        perform: gralloc_perform,
    },
    framebuffer: 0,
```

```
    fbFormat: 0,
    flags: 0,
    numBuffers: 0,
    bufferMask: 0,
    lock: PTHREAD_MUTEX_INITIALIZER,
    currentBuffer: 0,
    pmem_master: -1,
    pmem_master_base: 0,
};
```

上述代码和 Gralloc 中的代码基本一致，只是增加了结构体 private_module_t 的 perform()函数指针来实现 gralloc_perform()。函数 gralloc_perform()在文件 mapper.cpp 中定义，代码如下所示。

```
int gralloc_perform(struct gralloc_module_t const* module,
        int operation, ... )
{
    int res = -EINVAL;
    va_list args;
    va_start(args, operation);

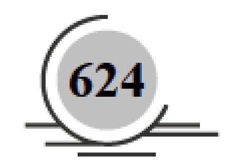
    switch (operation) {
        case GRALLOC_MODULE_PERFORM_CREATE_HANDLE_FROM_BUFFER: {
            int fd = va_arg(args, int);
            size_t size = va_arg(args, size_t);
            size_t offset = va_arg(args, size_t);
            void* base = va_arg(args, void*);
            native_handle_t** handle = va_arg(args, native_handle_t**);
            private_handle_t* hnd = (private_handle_t*)native_handle_create(
                    private_handle_t::sNumFds, private_handle_t::sNumInts);
            hnd->magic = private_handle_t::sMagic;
            hnd->fd = fd;
            hnd->flags = private_handle_t::PRIV_FLAGS_USES_PMEM;
            hnd->size = size;
            hnd->offset = offset;
            hnd->base = intptr_t(base) + offset;
            hnd->lockState = private_handle_t::LOCK_STATE_MAPPED;
            *handle = (native_handle_t *)hnd;
            res = 0;
            break;
        }
    }

    va_end(args);
    return res;
}
```

在上述代码中，通过 case 语句只实现了一个命令 GRALLOC_MODULE_PERFORM_CREATE_HANDLE_FROM_BUFFER，此命令是在 SurfaceFlinger 中被调用的内容，这是一个可选的功能，在此通过调用 Pmem 获取内存的大小。

18.7.2 文件 framebuffer.cpp

在文件 hardware/msm7k/libgralloc-qsd8k/framebuffer.cpp 中实现了 QSD8K 的 framebuffer_device_t 设备驱



动，其实现源码和标准的程序类似，区别是增加了更多颜色格式的支持，并且用 RGBA8888 作为默认的颜色格式。

其中在 post 中的区别是不支持双缓冲需要的内存复制功能时，不会再调用 memcopy()来实现，而是调用 msm_copy_buffer()来实现。Post 实现函数的代码如下所示。

```
static int fb_post(struct framebuffer_device_t* dev, buffer_handle_t buffer){
    if (private_handle_t::validate(buffer) < 0)
        return -EINVAL;
    int nxtIdx;
    bool reuse;
    struct qbuf_t qb;
    fb_context_t* ctx = (fb_context_t*)dev;
    private_handle_t const* hnd = reinterpret_cast<private_handle_t const*>(buffer);
    private_module_t* m = reinterpret_cast<private_module_t*>(
            dev->common.module);
    if (hnd->flags & private_handle_t::PRIV_FLAGS_FRAMEBUFFER) {
        reuse = false;
        nxtIdx = (m->currentIdx + 1) % NUM_BUFFERS;
        if (m->swapInterval == 0) {
            // if SwapInterval = 0 and no buffers available then reuse
            // current buf for next rendering so don't post new buffer
            if (pthread_mutex_trylock(&(m->avail[nxtIdx].lock))) {
                reuse = true;
            } else {
                if (! m->avail[nxtIdx].is_avail)
                    reuse = true;

                pthread_mutex_unlock(&(m->avail[nxtIdx].lock));
            }
        } else {      // swapInterval = 1
            if ((m->mddi_panel) && (m->currentIdx >= 0))   {
                // make sure prior posting of this buf is avail
                pthread_mutex_lock(&(m->avail[m->currentIdx].lock));
                if (! m->avail[m->currentIdx].is_avail) {
                        pthread_cond_wait(&(m->avail[m->currentIdx].cond),
                                        &(m->avail[m->currentIdx].lock));
                        m->avail[m->currentIdx].is_avail = true;
                    }
                    pthread_mutex_unlock(&(m->avail[m->currentIdx].lock));
            }
        }
        if(!reuse){
            // unlock previous ("current") Buffer and lock the new buffer
                if (m->currentBuffer && m->mddi_panel) {
                    m->base.unlock(&m->base, m->currentBuffer);
                }
                m->base.lock(&m->base, buffer,
                        private_module_t::PRIV_USAGE_LOCKED_FOR_POST,
                        0,0, m->info.xres, m->info.yres, NULL);
            pthread_mutex_lock(&(m->avail[nxtIdx].lock));
            m->avail[nxtIdx].is_avail = false;
```

```
                pthread_mutex_unlock(&(m->avail[nxtIdx].lock));
                qb.idx = nxtIdx;
                qb.buf = buffer;
                pthread_mutex_lock(&(m->qlock));
                m->disp.push(qb);
                pthread_cond_signal(&(m->qpost));
                pthread_mutex_unlock(&(m->qlock));
                // LCDC: after new buffer grabbed by MDP can unlock previous
                // (current) buffer
                if (!m->mddi_panel && m->currentBuffer) {
                    if (m->swapInterval != 0) {
                        pthread_mutex_lock(&(m->avail[m->currentIdx].lock));
                        if (! m->avail[m->currentIdx].is_avail) {
                            pthread_cond_wait(&(m->avail[m->currentIdx].cond),
                                             &(m->avail[m->currentIdx].lock));
                            m->avail[m->currentIdx].is_avail = true;
                        }
                        pthread_mutex_unlock(&(m->avail[m->currentIdx].lock));
                    }
                    m->base.unlock(&m->base, m->currentBuffer);
                }
                m->currentBuffer = buffer;
                m->currentIdx = nxtIdx;
            } else {
                if (m->currentBuffer)
                    m->base.unlock(&m->base, m->currentBuffer);
                m->base.lock(&m->base, buffer,
                             private_module_t::PRIV_USAGE_LOCKED_FOR_POST,
                             0,0, m->info.xres, m->info.yres, NULL);
                m->currentBuffer = buffer;
            }
        } else {
            void* fb_vaddr;
            void* buffer_vaddr;
            m->base.lock(&m->base, m->framebuffer,
                    GRALLOC_USAGE_SW_WRITE_RARELY,
                    0, 0, m->info.xres, m->info.yres,
                    &fb_vaddr);
            m->base.lock(&m->base, buffer,
                    GRALLOC_USAGE_SW_READ_RARELY,
                    0, 0, m->info.xres, m->info.yres,
                    &buffer_vaddr);
            //memcpy(fb_vaddr, buffer_vaddr, m->finfo.line_length * m->info.yres);
            msm_copy_buffer(
                    m->framebuffer, m->framebuffer->fd,
                    m->info.xres, m->info.yres, m->fbFormat,
                    m->info.xoffset, m->info.yoffset,
                    m->info.width, m->info.height);
            m->base.unlock(&m->base, buffer);
            m->base.unlock(&m->base, m->framebuffer);
        }
```

```
    return 0;
}
```

18.7.3　文件 gralloc.cpp

在文件 libgralloc-qsd8k/gralloc.cpp 中，实现了函数 alloc()、free()和 close()，这些函数是 MSM 的 Galloc 模块默认的实现函数，和标准函数是有一些区别的。函数 gralloc_alloc()是 MSM 中 gralloc_device_t 设备的分配函数。如果其参数不具有 GRALLOC_USAGE_HW_FB 宏，则会调用 gralloc_alloc_buffer 来分配内存。由此可见这部分的实现和默认 Galloc 模块中是不一样的。函数 Gallocgralloc_alloc_buffer()的实现代码如下所示。

```
static int gralloc_alloc_buffer(alloc_device_t* dev,
        size_t size, int usage, buffer_handle_t* pHandle)
{
    int err = 0;
    int flags = 0;
    int fd = -1;
    void* base = 0;
    int offset = 0;
    int lockState = 0;
    size = roundUpToPageSize(size);
   if (usage & GRALLOC_USAGE_HW_TEXTURE) {
        // enable pmem in that case, so our software GL can fallback to
        // the copybit module.
        flags |= private_handle_t::PRIV_FLAGS_USES_PMEM;
    }

    if (usage & GRALLOC_USAGE_HW_2D) {
        flags |= private_handle_t::PRIV_FLAGS_USES_PMEM;
    }

    if ((flags & private_handle_t::PRIV_FLAGS_USES_PMEM) == 0) {
try_ashmem:
        fd = ashmem_create_region("gralloc-buffer", size);
        if (fd < 0) {
            LOGE("couldn't create ashmem (%s)", strerror(errno));
            err = -errno;
        }
    } else {
        private_module_t* m = reinterpret_cast<private_module_t*>(
                dev->common.module);

        err = init_pmem_area(m);
        if (err == 0) {
            // PMEM buffers are always mmapped
            base = m->pmem_master_base;
            lockState |= private_handle_t::LOCK_STATE_MAPPED;

            offset = sAllocator.allocate(size);
            if (offset < 0) {
                // no more pmem memory
                err = -ENOMEM;
```

```
            } else {
                struct pmem_region sub = { offset, size };
                int openFlags = O_RDWR | O_SYNC;
                uint32_t uread = usage & GRALLOC_USAGE_SW_READ_MASK;
                uint32_t uwrite = usage & GRALLOC_USAGE_SW_WRITE_MASK;
                if (uread == GRALLOC_USAGE_SW_READ_OFTEN ||
                        uwrite == GRALLOC_USAGE_SW_WRITE_OFTEN) {
                        openFlags &= ~O_SYNC;
                }

                // now create the "sub-heap"
                fd = open("/dev/pmem", openFlags, 0);
                err = fd < 0 ? fd : 0;

                // and connect to it
                if (err == 0)
                        err = ioctl(fd, PMEM_CONNECT, m->pmem_master);

                // and make it available to the client process
                if (err == 0)
                        err = ioctl(fd, PMEM_MAP, &sub);

                if (err < 0) {
                       err = -errno;
                        close(fd);
                        sAllocator.deallocate(offset);
                        fd = -1;
                } else {
                        memset((char*)base + offset, 0, size);
                        // clean and invalidate the new allocation
                        cacheflush(intptr_t(base) + offset, size, 0);
                }
                //LOGD_IF(!err, "allocating pmem size=%d, offset=%d", size, offset);
            }
        } else {
            if ((usage & GRALLOC_USAGE_HW_2D) == 0) {
                // the caller didn't request PMEM, so we can try something else
                flags &= ~private_handle_t::PRIV_FLAGS_USES_PMEM;
                err = 0;
                goto try_ashmem;
            } else {
                LOGE("couldn't open pmem (%s)", strerror(errno));
            }
        }
    }

    if (err == 0) {
        private_handle_t* hnd = new private_handle_t(fd, size, flags);
        hnd->offset = offset;
        hnd->base = int(base)+offset;
        hnd->lockState = lockState;
```

```
            *pHandle = hnd;
        }

        LOGE_IF(err, "gralloc failed err=%s", strerror(-err));

        return err;
    }
```

在文件 gralloc.cpp 中，函数 init_pmem_area_locked()比较重要，它能够从默认的内存中实现内存映射功能，并且它是通过文件描述符实现的，这和 mapBuffer()有很大的不同。函数 init_pmem_area_locked()的实现代码如下所示。

```
static int init_pmem_area_locked(private_module_t* m)
 {
        int err = 0;
        int master_fd = open("/dev/pmem", O_RDWR, 0);
        if (master_fd >= 0) {
            size_t size;
            pmem_region region;
            if (ioctl(master_fd, PMEM_GET_TOTAL_SIZE, &region) < 0) {
                LOGE("PMEM_GET_TOTAL_SIZE failed, limp mode");
                size = 8<<20;      // 8 MiB
            } else {
                size = region.len;
            }
            sAllocator.setSize(size);

            void* base = mmap(0, size,
                    PROT_READ|PROT_WRITE, MAP_SHARED, master_fd, 0);
            if (base == MAP_FAILED) {
                err = -errno;
                base = 0;
                close(master_fd);
                master_fd = -1;
            }
            m->pmem_master = master_fd;
            m->pmem_master_base = base;
        } else {
            err = -errno;
        }
        return err;
 }
```

上述函数的运行流程如下。

☑　通过 open("/dev/pmem", O_RDWR, 0)打开 PMEM。

☑　通过 ioctl(master_fd, PMEM_GET_TOTAL_SIZE, &region)获取所有空间。

☑　分配 sAllocator.setSize(size)指定数目的空间。

☑　完成映射功能。

再看函数 gralloc_free()，此函数用于实现 alloc_device_t 中的释放功能，和默认的 Gralloc 相比，区别是在此不使用 PRIV_FLAGS_FRAMEBUFFER 标志实现。函数 gralloc_free()的实现代码如下所示。

```
static int gralloc_free(alloc_device_t* dev,
            buffer_handle_t handle)
```

```
{
    if (private_handle_t::validate(handle) < 0)
        return -EINVAL;

    private_handle_t const* hnd = reinterpret_cast<private_handle_t const*>(handle);
    if (hnd->flags & private_handle_t::PRIV_FLAGS_FRAMEBUFFER) {
        // free this buffer
        private_module_t* m = reinterpret_cast<private_module_t*>(
                dev->common.module);
        const size_t bufferSize = m->finfo.line_length * m->info.yres;
        int index = (hnd->base - m->framebuffer->base) / bufferSize;
        m->bufferMask &= ~(1<<index);
    } else {
        if (hnd->flags & private_handle_t::PRIV_FLAGS_USES_PMEM) {
            if (hnd->fd >= 0) {
                struct pmem_region sub = { hnd->offset, hnd->size };
                int err = ioctl(hnd->fd, PMEM_UNMAP, &sub);
                LOGE_IF(err<0, "PMEM_UNMAP failed (%s), "
                        "fd=%d, sub.offset=%lu, sub.size=%lu",
                        strerror(errno), hnd->fd, hnd->offset, hnd->size);
               if (err == 0) {
                    // we can't deallocate the memory in case of UNMAP failure
                    // because it would give that process access to someone else's
                    // surfaces, which would be a security breach
                    sAllocator.deallocate(hnd->offset);
                }
            }
        }

        gralloc_module_t* module = reinterpret_cast<gralloc_module_t*>(
                dev->common.module);
        terminateBuffer(module, const_cast<private_handle_t*>(hnd));
    }

    close(hnd->fd);
    delete hnd;
    return 0;
}
```

由此可见，在 MSM 平台中提高了 alloc_device_t 设备的性能，使用 Pmem 驱动程序作为内存映射工具，将原来通过 ashmem 分配和管理的内存部分转移到了 Pmem 上面。

18.8　OMAP 处理器中的显示驱动实现

本节将详细剖析 OMAP 处理器平台显示系统驱动的实现过程。OMAP 平台的驱动程序由 FrameBuffer 驱动和 Gralloc 模块构成，其中里面的 FrameBuffer 是标准驱动，而 Gralloc 模块既可以使用默认的，也可以使用自己自定义的。

18.8.1　文件 omapfb-main.c

OMAP 处理器中 FrameBuffer 驱动的主要实现文件是 drivers/video/omap2/omapfb/omapfb-main.c，在此文件中通过函数 omapfb_create_framebuffers()来注册 FrameBuffer 驱动程序，此函数的实现代码如下所示。

```
static int omapfb_create_framebuffers(struct omapfb2_device *fbdev)
{
	int r, i;
	fbdev->num_fbs = 0;
	DBG("create %d framebuffers\n", CONFIG_FB_OMAP2_NUM_FBS);
	/* allocate fb_infos */
	for (i = 0; i < CONFIG_FB_OMAP2_NUM_FBS; i++) {
		struct fb_info *fbi;
		struct omapfb_info *ofbi;
		fbi = framebuffer_alloc(sizeof(struct omapfb_info),
				fbdev->dev);
		if (fbi == NULL) {
			dev_err(fbdev->dev,
				"unable to allocate memory for plane info\n");
			return -ENOMEM;
		}
		clear_fb_info(fbi);
		fbdev->fbs[i] = fbi;
		ofbi = FB2OFB(fbi);
		ofbi->fbdev = fbdev;
		ofbi->id = i;
		ofbi->region = &fbdev->regions[i];
		ofbi->region->id = i;
		init_rwsem(&ofbi->region->lock);
		/* assign these early, so that fb alloc can use them */
		ofbi->rotation_type = def_vrfb ? OMAP_DSS_ROT_VRFB :
			OMAP_DSS_ROT_DMA;
		ofbi->mirror = def_mirror;
		fbdev->num_fbs++;
	}
	DBG("fb_infos allocated\n");
	/* assign overlays for the fbs */
	for (i = 0; i < min(fbdev->num_fbs, fbdev->num_overlays); i++) {
		struct omapfb_info *ofbi = FB2OFB(fbdev->fbs[i]);
		ofbi->overlays[0] = fbdev->overlays[i];
		ofbi->num_overlays = 1;
	}
	/* allocate fb memories */
	r = omapfb_allocate_all_fbs(fbdev);
	if (r) {
		dev_err(fbdev->dev, "failed to allocate fbmem\n");
		return r;
	}
	DBG("fbmems allocated\n");
	/* setup fb_infos */
```

```
		for (i = 0; i < fbdev->num_fbs; i++) {
			struct fb_info *fbi = fbdev->fbs[i];
			struct omapfb_info *ofbi = FB2OFB(fbi);
			omapfb_get_mem_region(ofbi->region);
			r = omapfb_fb_init(fbdev, fbi);
			omapfb_put_mem_region(ofbi->region);
			if (r) {
				dev_err(fbdev->dev, "failed to setup fb_info\n");
				return r;
			}
		}
		DBG("fb_infos initialized\n");
		for (i = 0; i < fbdev->num_fbs; i++) {
			r = register_framebuffer(fbdev->fbs[i]);
			if (r != 0) {
				dev_err(fbdev->dev,
					"registering framebuffer %d failed\n", i);
				return r;
			}
		}

		DBG("framebuffers registered\n");
		for (i = 0; i < fbdev->num_fbs; i++) {
			struct fb_info *fbi = fbdev->fbs[i];
			struct omapfb_info *ofbi = FB2OFB(fbi);

			omapfb_get_mem_region(ofbi->region);
			r = omapfb_apply_changes(fbi, 1);
			omapfb_put_mem_region(ofbi->region);
			if (r) {
				dev_err(fbdev->dev, "failed to change mode\n");
				return r;
			}
		}
		/* Enable fb0 */
		if (fbdev->num_fbs > 0) {
			struct omapfb_info *ofbi = FB2OFB(fbdev->fbs[0]);
			if (ofbi->num_overlays > 0) {
				struct omap_overlay *ovl = ofbi->overlays[0];
				r = omapfb_overlay_enable(ovl, 1);
				if (r) {
					dev_err(fbdev->dev,
							"failed to enable overlay\n");
					return r;
				}
			}
		}
		DBG("create_framebuffers done\n");
		return 0;
}
```

18.8.2　文件 omapfb.h

在文件 include/linux/omapfb.h 中定义了额外的 ioctl 命令号，具体代码如下所示。

```
#define OMAP_IOW(num, dtype)      _IOW('O', num, dtype)
#define OMAP_IOR(num, dtype)      _IOR('O', num, dtype)
#define OMAP_IOWR(num, dtype)     _IOWR('O', num, dtype)
#define OMAP_IO(num)              _IO('O', num)
#define OMAPFB_MIRROR             OMAP_IOW(31, int)
#define OMAPFB_SYNC_GFX           OMAP_IO(37)
#define OMAPFB_VSYNC              OMAP_IO(38)
#define OMAPFB_SET_UPDATE_MODE    OMAP_IOW(40, int)
#define OMAPFB_GET_CAPS           OMAP_IOR(42, struct omapfb_caps)
#define OMAPFB_GET_UPDATE_MODE    OMAP_IOW(43, int)
#define OMAPFB_LCD_TEST           OMAP_IOW(45, int)
#define OMAPFB_CTRL_TEST          OMAP_IOW(46, int)
#define OMAPFB_UPDATE_WINDOW_OLD OMAP_IOW(47, struct omapfb_update_window_old)
#define OMAPFB_SET_COLOR_KEY      OMAP_IOW(50, struct omapfb_color_key)
#define OMAPFB_GET_COLOR_KEY      OMAP_IOW(51, struct omapfb_color_key)
#define OMAPFB_SETUP_PLANE        OMAP_IOW(52, struct omapfb_plane_info)
#define OMAPFB_QUERY_PLANE        OMAP_IOW(53, struct omapfb_plane_info)
#define OMAPFB_UPDATE_WINDOW      OMAP_IOW(54, struct omapfb_update_window)
#define OMAPFB_SETUP_MEM          OMAP_IOW(55, struct omapfb_mem_info)
#define OMAPFB_QUERY_MEM          OMAP_IOW(56, struct omapfb_mem_info)
#define OMAPFB_WAITFORVSYNC       OMAP_IO(57)
#define OMAPFB_MEMORY_READ        OMAP_IOR(58, struct omapfb_memory_read)
#define OMAPFB_GET_OVERLAY_COLORMODE OMAP_IOR(59, struct omapfb_ovl_colormode)
#define OMAPFB_WAITFORGO          OMAP_IO(60)
#define OMAPFB_GET_VRAM_INFO      OMAP_IOR(61, struct omapfb_vram_info)
#define OMAPFB_SET_TEARSYNC       OMAP_IOW(62, struct omapfb_tearsync_info)
#define OMAPFB_GET_DISPLAY_INFO OMAP_IOR(63, struct omapfb_display_info)
```

18.9　实 战 演 练

通过本章前面内容的详细讲解，读者已经基本了解了 Android 系统中 LCD 驱动的核心架构知识和具体实现源码，本节将通过具体实例来讲解在 Android 系统中移植 LCD 驱动的基本知识。

18.9.1　S3C2440 上的 LCD 驱动

要想使一块 LCD 能够正常显示文字和图像，不仅需要 LCD 驱动器，而且还需要相应的 LCD 控制器。在大多数情况下，生产厂商会把 LCD 驱动器以 COF/COG 的形式与 LCD 玻璃基板制作在一起。而 LCD 控制器则是由外部的电路来实现的，现在很多的 MCU 内部都集成了 LCD 控制器，例如 S3C2410/2440 等。通过使用 LCD 控制器就可以产生 LCD 驱动器所需要的控制信号，这样来控制 STN/TFT 屏的显示工作。

S3C2440 内部 LCD 控制器结构如图 18-4 所示。

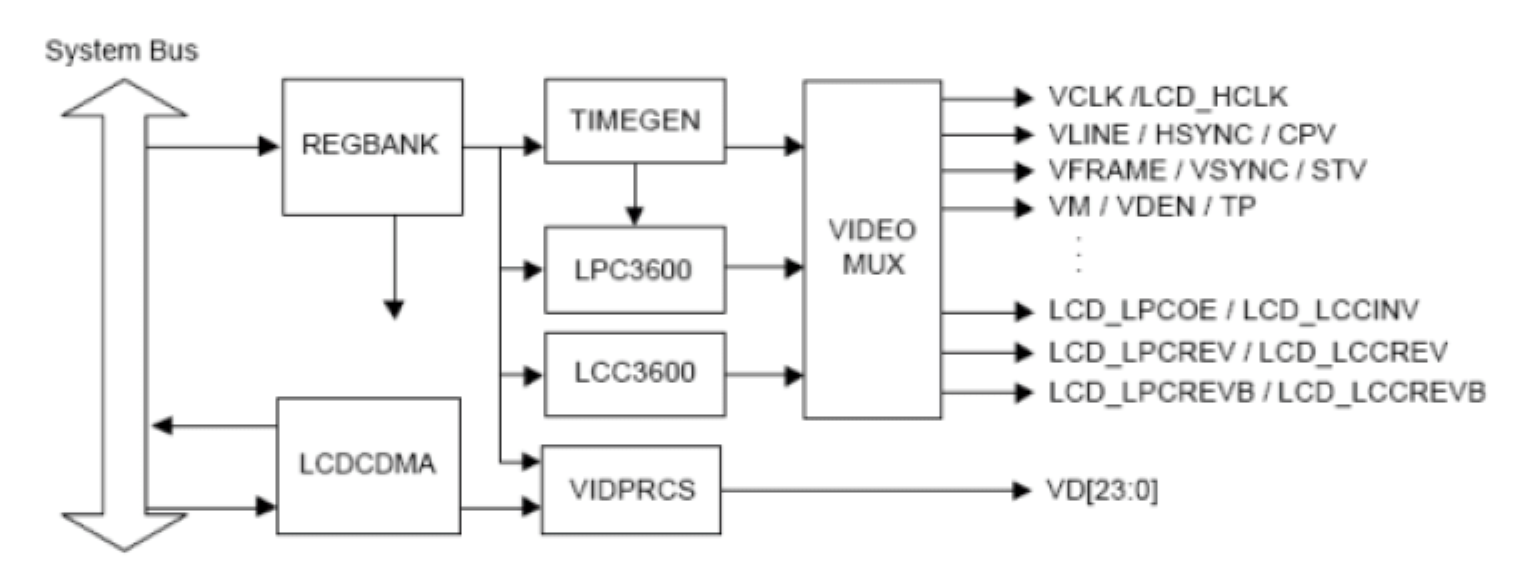


图 18-4　S3C2440 内部 LCD 控制器结构

在 S3C2440 的数据手册中描述了这个集成在 S3C2440 内部的 LCD 控制器，具体描述如下。

☑ LCD 控制器由 REGBANK、LCDCDMA、TIMEGEN、VIDPRCS 寄存器组成。

☑ REGBANK 由 17 个可编程的寄存器组和一块 256×16 的调色板内存组成，它们用来配置 LCD 控制器。

☑ LCDCDMA 是一个专用的 DMA，它能自动地把在帧内存中的视频数据传送到 LCD 驱动器，通过使用这个 DMA 通道，视频数据在不需要 CPU 干预的情况下显示在 LCD 屏上。

☑ VIDPRCS 接收来自 LCDCDMA 的数据，将数据转换为合适的数据格式，例如 4/8 位单扫，4 位双扫显示模式，然后通过数据端口 VD[23:0]传送视频数据到 LCD 驱动器。

☑ TIMEGEN 由可编程的逻辑组成，它生成 LCD 驱动器需要的控制信号，例如 VSYNC、HSYNC、VCLK 和 LEND 等，而这些控制信号又与 REGBANK 寄存器组中的 LCDCON1/2/3/4/5 的配置密切相关，通过不同的配置，TIMEGEN 就能产生这些信号的不同形态，从而支持不同的 LCD 驱动器（即不同的 STN/TFT 屏）。

在现实应用中，常见 TFT 屏的工作时序图如图 18-5 所示。

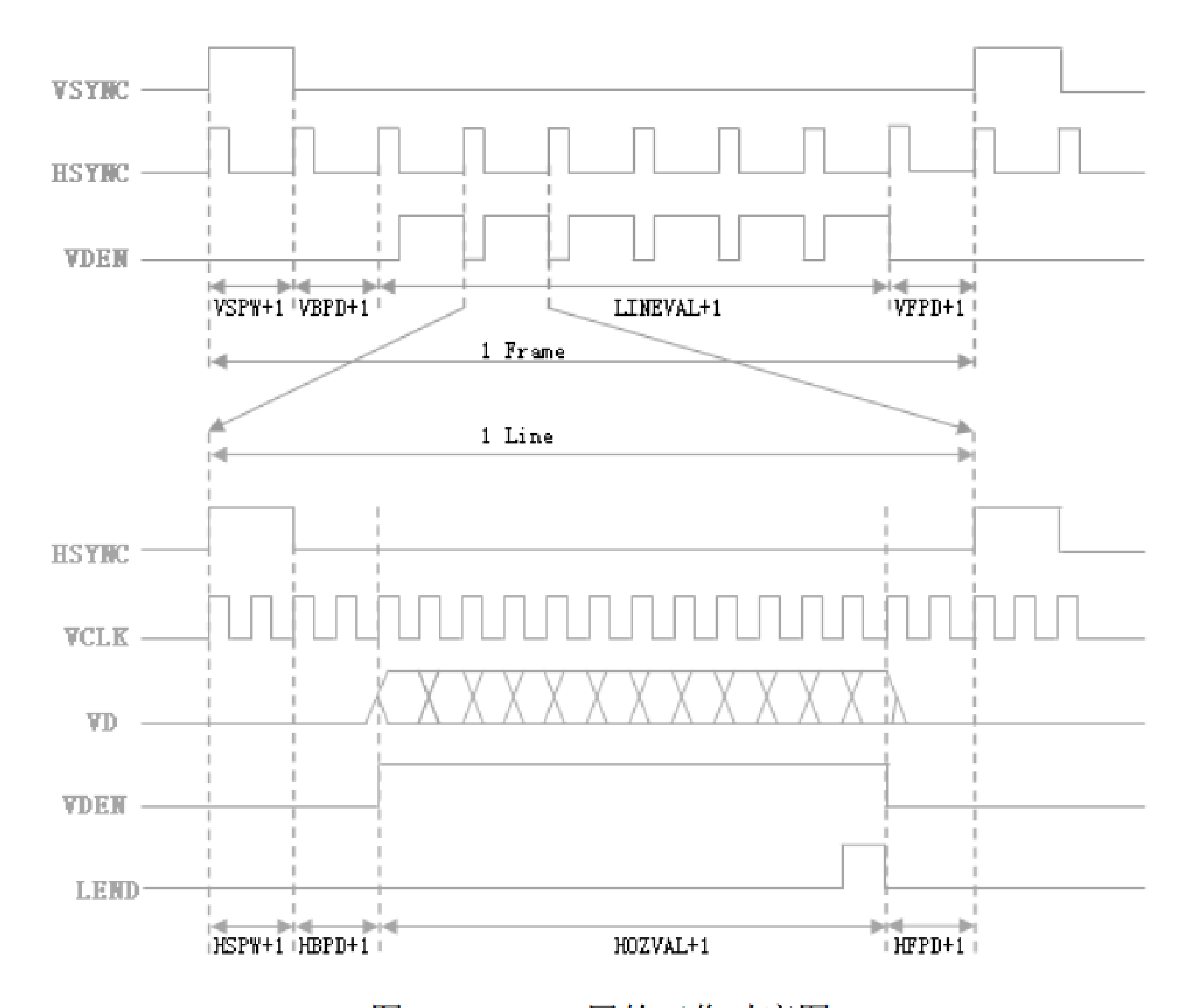


图 18-5　TFT 屏的工作时序图

LCD 提供了如下的外部接口信号。

☑ VSYNC/VFRAME/STV：垂直同步信号（TFT）/帧同步信号（STN）/SEC TFT 信号。
☑ HSYNC/VLINE/CPV：水平同步信号（TFT）/行同步脉冲信号（STN）/SEC TFT 信号。
☑ VCLK/LCD_HCLK：像素时钟信号（TFT/STN）/SEC TFT 信号。
☑ VD[23:0]：LCD 像素数据输出端口（TFT/STN/SEC TFT）。
☑ VDEN/VM/TP：数据使能信号（TFT）/LCD 驱动交流偏置信号（STN）/SEC TFT 信号。
☑ LEND/STH：行结束信号（TFT）/SEC TFT 信号。
☑ LCD_LPCOE：SEC TFT OE 信号。
☑ LCD_LPCREV：SEC TFT REV 信号。
☑ LCD_LPCREVB：SEC TFT REVB 信号。

所有显示器显示图像的原理都是从上到下，从左到右。举个例子，一幅图像可以看作是一个矩形，由很多排列整齐的点一行一行组成，这些点称为像素。那么这幅图在 LCD 上的显示原理如下。

☑ A：显示指针从矩形左上角的第一行第一个点开始，一个点一个点地在 LCD 上显示，在上面的时序图上用时间线表示就是 VCLK，我们称之为像素时钟信号。
☑ B：当显示指针一直显示到矩形的右边就结束这一行，那么这一行的动作在上面的时序图中就称之为 1 Line。
☑ C：接下来显示指针又回到矩形的左边从第二行开始显示，注意，显示指针在从第一行的右边回到第二行的左边是需要一定时间的，我们称之为行切换。
☑ D：依次类推，显示指针就这样一行一行地显示至矩形的右下角，这样才逐渐将一幅图显示完整。因此，这一行一行的显示在时间线上看，就是时序图上的 HSYNC。
☑ E：然而，LCD 的显示并不是对一幅图像快速地显示一下，为了持续和稳定地在 LCD 上显示，就需要切换到另一幅图（另一幅图可以和上一幅图一样或者不一样，目的只是为了将图像持续地显示在 LCD 上）上。那么这一幅一幅的图像就称之为帧，在时序图上就表示为 1 Frame，因此从时序图上可以看出 1 Line 只是 1 Frame 中的一行。
☑ F：同样地，在帧与帧切换之间也是需要一定时间的，我们称之为帧切换，那么 LCD 整个显示的过程在时间线上看，就可表示为时序图上的 VSYNC。

上面时序图中各个时钟延时参数的含义如下。

☑ VBPD（Vertical Back Porch）：表示在一帧图像开始时，垂直同步信号以后的无效的行数，对应驱动中的 upper_margin。
☑ VFBD（Vertical Front Porch）：表示在一帧图像结束后，垂直同步信号以前的无效的行数，对应驱动中的 lower_margin。
☑ VSPW（Vertical Sync Pulse Width）：表示垂直同步脉冲的宽度，用行数计算，对应驱动中的 vsync_len。
☑ HBPD（Horizontal Back Porch）：表示从水平同步信号开始到一行的有效数据开始之间的 VCLK 的个数，对应驱动中的 left_margin。
☑ HFPD（Horizontal Front Porth）：表示一行的有效数据结束到下一个水平同步信号开始之间的 VCLK 的个数，对应驱动中的 right_margin。
☑ HSPW（Horizontal Sync Pulse Width）：表示水平同步信号的宽度，用 VCLK 计算，对应驱动中的 hsync_len。

1．帧缓冲（FrameBuffer）

帧缓冲是 Linux 为显示设备提供的一个接口，它把一些显示设备描述成一个缓冲区，允许应用程序通过

FrameBuffer 定义好的接口访问这些图形设备，从而不用去关心具体的硬件细节。对于帧缓冲设备而言，只要在显示缓冲区与显示点对应的区域写入颜色值，对应的颜色就会自动地在屏幕上显示。帧缓冲设备为标准的字符型设备，在 Linux 中主设备号为 29，定义在/include/linux/major.h 中的 FB_MAJOR，次设备号定义帧缓冲的个数，最大允许有 32 个 FrameBuffer，定义在/include/linux/fb.h 中的 FB_MAX，对应于文件系统下/dev/fb%d 设备文件。

帧缓冲设备在 Linux 中也可以看作是一个完整的子系统，基本由 fbmem.c 和 xxxfb.c 组成。向上给应用程序提供完善的设备文件操作接口（即对 FrameBuffer 设备进行 read、write、ioctl 等操作），接口在 Linux 提供的 fbmem.c 文件中实现；向下提供了硬件操作的接口，只是这些接口 Linux 并没有提供实现，因为要根据具体的 LCD 控制器硬件进行设置，所以这就是我们要做的事情（即 xxxfb.c 部分的实现）。

从帧缓冲设备驱动程序结构看，该驱动主要和 fb_info 结构体有关，该结构体记录了帧缓冲设备的全部信息，包括设备的设置参数、状态以及对底层硬件操作的函数指针。在 Linux 中，每一个帧缓冲设备都必须对应一个 fb_info，fb_info 在文件/linux/fb.h 中的定义如下所示。

```
struct fb_info{
    int node;
    int flags;
    struct fb_var_screeninfo var;              /*LCD 可变参数结构体*/
    struct fb_fix_screeninfo fix;              /*LCD 固定参数结构体*/
    struct fb_monspecs monspecs;               /*LCD 显示器标准*/
    struct work_struct queue;                  /*帧缓冲事件队列*/
    struct fb_pixmap pixmap;                   /*图像硬件 mapper* /
    struct  fb_pixmap sprite;                  /*光标硬件 mapper*/
    struct fb_cmap cmap;                       /*当前的颜色表*/
    struct fb_videomode *mode;                 /*当前的显示模式*/

#ifdef CONFIG_FB_BACKLIGHT
    struct backlight_device*bl_dev;            /*对应的背光设备*/
    struct mutex bl_curve_mutex;
    u8 bl_curve[FB_BACKLIGHT_LEVELS];          /*背光调整*/
#endif
#ifdef CONFIG_FB_DEFERRED_IO
    struct delayed_work deferred_work;
    struct fb_deferred_io *fbdefio;
#endif

    struct fb_ops *fbops;                      /*对底层硬件操作的函数指针*/
    struct device *device;
    struct device *dev;                        /*fb 设备*/
    int class_flag;
#ifdef CONFIG_FB_TILEBLITTING
    struct fb_tile_ops *tileops;               /*图块 Blitting* /
#endif
    char __iomem *screen_base;                 /*虚拟基地址*/
    unsignedlong screen_size;                  /*LCD 的 I/O 映射的虚拟内存大小*/
    void*pseudo_palette;                       /*伪 16 色颜色表*/
#define   FBINFO_STATE_RUNNING      0
#define FBINFO_STATE_SUSPENDED   1
    u32 state;                                 /*LCD 的挂起或恢复状态*/
    void*fbcon_par;
```

```
    void*par;
};
```

在上述代码中，比较重要的成员有 struct fb_var_screeninfo var、struct fb_fix_screeninfo fix 和 struct fb_ops*fbops，它们都是结构体。

（1）fb_var_screeninfo 结构体主要记录用户可以修改的控制器的参数，例如屏幕的分辨率和每个像素的比特数等，定义该结构体的代码如下所示。

```
struct fb_var_screeninfo{
    __u32 xres;                         /*可见屏幕一行有多少个像素点*/
    __u32 yres;                         /*可见屏幕一列有多少个像素点*/
    __u32 xres_virtual;                 /*虚拟屏幕一行有多少个像素点*/
    __u32 yres_virtual;                 /*虚拟屏幕一列有多少个像素点*/
    __u32 xoffset;                      /*虚拟到可见屏幕之间的行偏移*/
    __u32 yoffset;                      /*虚拟到可见屏幕之间的列偏移*/
    __u32 bits_per_pixel;               /*每个像素的位数即 BPP*/
    __u32 grayscale;                    /*非 0 时，指的是灰度*/

    struct fb_bitfield red;             /*fb 缓存的 R 位域*/
    struct fb_bitfield green;           /*fb 缓存的 G 位域*/
    struct fb_bitfield blue;            /*fb 缓存的 B 位域*/
    struct fb_bitfield transp;          /*透明度*/

    __u32 nonstd;                       /* != 0  非标准像素格式*/
    __u32 activate;
    __u32 height;                       /*高度*/
    __u32 width;                        /*宽度*/
    __u32 accel_flags;

    /*定时：除了 pixclock 本身外，其他的都以像素时钟为单位*/
    __u32 pixclock;                     /*像素时钟（皮秒）*/
    __u32 left_margin;                  /*行切换，从同步到绘图之间的延迟*/
    __u32 right_margin;                 /*行切换，从绘图到同步之间的延迟*/
    __u32 upper_margin;                 /*帧切换，从同步到绘图之间的延迟*/
    __u32 lower_margin;                 /*帧切换，从绘图到同步之间的延迟*/
    __u32 hsync_len;                    /*水平同步的长度*/
    __u32 vsync_len;                    /*垂直同步的长度*/
    __u32 sync;
    __u32 vmode;
    __u32 rotate;
    __u32 reserved[5];                  /*保留*/
};
```

（2）fb_fix_screeninfo 结构体主要用于记录用户不可以修改的控制器的参数，例如屏幕缓冲区的物理地址和长度等，定义该结构体的代码如下所示。

```
struct fb_fix_screeninfo{
    char id[16];                        /*字符串形式的标识符 */
    unsignedlong smem_start;            /*fb 缓存的开始位置 */
    __u32 smem_len;                     /*fb 缓存的长度 */
    __u32 type;                         /*看 FB_TYPE_* */
    __u32 type_aux;                     /*分界*/
    __u32 visual;                       /*看 FB_VISUAL_* */
```

```
    __u16   xpanstep;                          /*如果没有硬件 panning 就赋值为 0 */
    __u16 ypanstep;                            /*如果没有硬件 panning 就赋值为 0 */
    __u16 ywrapstep;                           /*如果没有硬件 ywrap 就赋值为 0 */
    __u32 line_length;                         /*一行的字节数 */
    unsignedlong mmio_start;                   /*内存映射 I/O 的开始位置*/
    __u32 mmio_len;                            /*内存映射 I/O 的长度*/
    __u32 accel;
    __u16 reserved[3];                         /*保留*/
};
```

（3）fb_ops 结构体是对底层硬件操作的函数指针，该结构体中定义了对硬件的操作。定义该结构体的主要实现代码如下所示。

```
struct fb_ops{
    struct module *owner;
    //检查可变参数并进行设置
    int(*fb_check_var)(struct   fb_var_screeninfo*var,struct fb_info*info);
    //根据设置的值进行更新，使之有效
    int(*fb_set_par)(struct   fb_info*info);
    //设置颜色寄存器
    int(*fb_setcolreg)(unsigned   regno,unsigned red,unsigned green,
            unsigned blue,unsigned transp,struct fb_info*info);
    //显示空白
    int(*fb_blank)(int blank,struct fb_info *info);
    //矩形填充
    void(*fb_fillrect)(struct fb_info*info,conststruct fb_fillrect*rect);
    //复制数据
    void(*fb_copyarea)(struct fb_info*info,conststruct fb_copyarea*region);
    //图形填充
    void(*fb_imageblit)(struct fb_info*info,conststruct fb_image*image);
};
```

2．帧缓冲设备作为平台设备

在 S3C2440 中，LCD 控制器被集成在芯片的内部作为一个相对独立的单元，Linux 把它看作是一个平台设备，所以在内核代码文件/arch/arm/plat-s3c24xx/devs.c 中定义了和 LCD 相关的平台设备及资源，具体代码如下所示。

```
/* LCD Controller */
//LCD 控制器的资源信息
staticstruct resource s3c_lcd_resource[]={
    [0]={
        .start = S3C24XX_PA_LCD,                        //控制器 I/O 端口开始地址
        .end = S3C24XX_PA_LCD + S3C24XX_SZ_LCD -1,      //控制器 I/O 端口结束地址
        .flags= IORESOURCE_MEM,//标识为 LCD 控制器 I/O 端口，在驱动中引用这个就表示引用 I/O 端口
    },
    [1]={
        .start = IRQ_LCD,                               //LCD 中断
        .end = IRQ_LCD,
        .flags = IORESOURCE_IRQ,                        //标识为 LCD 中断
    }
};
```

```
static u64 s3c_device_lcd_dmamask = 0xffffffffUL;

struct platform_device   s3c_device_lcd ={
      .name             ="s3c2410-lcd",                          //作为平台设备的 LCD 设备名
      .id           =-1,
      .num_resources = ARRAY_SIZE(s3c_lcd_resource),        //资源数量
      .resource      = s3c_lcd_resource,                       //引用上面定义的资源
      .dev ={
            .dma_mask =&s3c_device_lcd_dmamask,
            .coherent_dma_mask = 0xffffffffUL
      }
};

//导出定义的 LCD 平台设备，以便在 mach-smdk2440.c 的 smdk2440_devices[]中添加到平台设备列表中
EXPORT_SYMBOL(s3c_device_lcd);
```

除此之外，Linux 还在文件/arch/arm/mach-s3c2410/include/mach/fb.h 中为 LCD 平台设备定义了一个 s3c2410fb_mach_info 结构体，该结构体主要是记录 LCD 的硬件参数信息（例如该结构体的 s3c2410fb_display 成员结构中就用于记录 LCD 的屏幕尺寸、屏幕信息、可变的屏幕参数、LCD 配置寄存器等），这样在写驱动时就直接使用这个结构体。下面来看一下内核是如何使用这个结构体的。在文件/arch/arm/mach-s3c2440/mach-smdk2440.c 中的定义代码如下所示。

```
/* LCD driver info */
//LCD 硬件的配置信息，注意这里使用的 LCD 是 NEC 3.5 寸 TFT 屏，这些参数要根据具体的 LCD 屏进行设置
staticstruct s3c2410fb_display smdk2440_lcd_cfg __initdata={

   //这个地方的设置是配置 LCD 寄存器 5，这些宏定义在 regs-lcd.h 中，计算后二进制为：111111111111，然后对照数据手册上 LCDCON5 的各位来看，注意是从右边开始
      .lcdcon5 = S3C2410_LCDCON5_FRM565|
                  S3C2410_LCDCON5_INVVLINE |
                   S3C2410_LCDCON5_INVVFRAME |
                  S3C2410_LCDCON5_PWREN |
                  S3C2410_LCDCON5_HWSWP,
      .type      = S3C2410_LCDCON1_TFT,//TFT 类型
      /* NEC 3.5" */
      .width = 240,//屏幕宽度
      .height = 320,//屏幕高度
   //以下一些参数在上面的时序图分析中讲到过，各参数的值请根据具体的 LCD 屏数据手册结合上面时序分析来设定
      .pixclock = 100000,//像素时钟
      .xres = 240,//水平可见的有效像素
      .yres = 320,//垂直可见的有效像素
      .bpp = 16,//色位模式
      .left_margin = 19,//行切换，从同步到绘图之间的延迟
      .right_margin= 36,//行切换，从绘图到同步之间的延迟
      .hsync_len = 5,//水平同步的长度
      .upper_margin= 1,//帧切换，从同步到绘图之间的延迟
      .lower_margin= 5,//帧切换，从绘图到同步之间的延迟
      .vsync_len = 1,//垂直同步的长度
};

staticstruct s3c2410fb_mach_info smdk2440_fb_info __initdata={
```

```
    .displays =&smdk2440_lcd_cfg,//应用上面定义的配置信息
    .num_displays = 1,
    .default_display = 0,
    //将 GPC0、GPC1 配置成 LEND 和 VCLK，将 GPC8-15 配置成 VD0-7，其他配置成普通输出 I/O 口
    .gpccon= 0xaaaa555a,
    .gpccon_mask = 0xffffffff,
    .gpcup = 0x0000ffff,//禁止 GPIOC 的上拉功能
    .gpcup_mask = 0xffffffff,
    .gpdcon = 0xaaaaaaaa,//将 GPD0-15 配置成 VD8-23
    .gpdcon_mask = 0xffffffff,
    .gpdup = 0x0000ffff,//禁止 GPIOD 的上拉功能
    .gpdup_mask = 0xffffffff,
    .lpcsel = 0x0,//这个是三星 TFT 屏的参数，这里不用
};
```

要使 LCD 控制器支持其他的 LCD 屏，需要根据 LCD 的数据手册修改以上这些参数的值。为了在驱动中引用到 s3c2410fb_mach_info 结构体，请看在文件 mach-smdk2440.c 中的如下代码。

```
//S3C2440 初始化函数
staticvoid __init  smdk2440_machine_init(void)
{
    //调用该函数将上面定义的 LCD 硬件信息保存到平台数据中
    s3c24xx_fb_set_platdata(&smdk2440_fb_info);
    s3c_i2c0_set_platdata(NULL);
    platform_add_devices(smdk2440_devices, ARRAY_SIZE(smdk2440_devices));
    smdk_machine_init();
}
s3c24xx_fb_set_platdata 定义在 plat-s3c24xx/devs.c 中：
void __init s3c24xx_fb_set_platdata(struct s3c2410fb_mach_info*pd)
{
    struct s3c2410fb_mach_info *npd;
    npd = kmalloc(sizeof(*npd), GFP_KERNEL);
    if(npd){
        memcpy(npd, pd,sizeof(*npd));
      //这里就是将内核中定义的 s3c2410fb_mach_info 结构体数据保存到 LCD 平台数据中，所以在写驱动时
就可以直接在平台数据中获取 s3c2410fb_mach_info 结构体的数据（即 LCD 各种参数信息）进行操作
        s3c_device_lcd.dev.platform_data= npd;
    }else{
        printk(KERN_ERR "no memory for LCD platform data/n");
    }
}
```

3．帧缓冲（FrameBuffer）设备驱动实例代码

（1）创建驱动文件 my2440_lcd.c，即创建驱动程序的最基本结构，实现 FrameBuffer 驱动的初始化和卸载部分功能，具体实现代码如下所示。

```
#include<linux/kernel.h>
#include<linux/module.h>
#include<linux/errno.h>
#include<linux/init.h>
#include<linux/platform_device.h>
#include<linux/dma-mapping.h>
#include<linux/fb.h>
```

```
#include<linux/clk.h>
#include<linux/interrupt.h>
#include<linux/mm.h>

#include<linux/slab.h>
#include<linux/delay.h>
#include<asm/irq.h>
#include<asm/io.h>
#include<asm/div64.h>
#include<mach/regs-lcd.h>
#include<mach/regs-gpio.h>
#include<mach/fb.h>
#include<linux/pm.h>
/*FrameBuffer 设备名称*/
static char driver_name[] ="my2440_lcd";
/*定义一个结构体用来维护驱动程序中各函数中用到的变量。先别看结构体要定义这些成员，到各函数使用的地方
就明白了*/
struct   my2440fb_var
{
      int lcd_irq_no;                              /*保存 LCD 中断号*/
      struct clk *lcd_clock;                       /*保存从平台时钟队列中获取的 LCD 时钟*/
      struct resource *lcd_mem;                    /*LCD 的 I/O 空间*/
      void __iomem *lcd_base;                      /*LCD 的 I/O 空间映射到虚拟地址*/
      struct device *dev;
      struct s3c2410fb_hw regs;/*表示 5 个 LCD 配置寄存器，s3c2410fb_hw 定义在 mach-s3c2410/include/mach/
fb.h 中* /
      /*定义一个数组来充当调色板。根据数据手册描述，TFT 屏色位模式为 8BPP 时，调色板（颜色表）的长度为
256，调色板起始地址为 0x4D000400*/
      u32      palette_buffer[256];
      u32 pseudo_pal[16];
      unsignedint palette_ready;/*标识调色板是否准备好了*/
};
/*用作清空调色板（颜色表）*/
#define PALETTE_BUFF_CLEAR(0x80000000)
/*LCD 平台驱动结构体，平台驱动结构体定义在 platform_device.h 中，该结构体成员接口函数在下面的第（2）
步中实现*/
staticstruct platform_driver lcd_fb_driver=
{
      .probe        = lcd_fb_probe,                     /*FrameBuffer 设备探测* /
      .remove       = __devexit_p(lcd_fb_remove),       /*FrameBuffer 设备移除*/
      .suspend      = lcd_fb_suspend,                   /*FrameBuffer 设备挂起*/
      .resume       = lcd_fb_resume,                    /*FrameBuffer 设备恢复*/
      .driver       =
      {
            /*注意这里的名称一定要和系统中定义平台设备的地方一致，这样才能把平台设备与该平台设备的驱动关
联起来*/
            .name ="s3c2410-lcd",
            .owner = THIS_MODULE,
      },
};
staticint __init lcd_init(void)
```

```
{
    /*在 Linux 中，帧缓冲设备被看作是平台设备，所以这里注册平台设备*/
    return platform_driver_register(&lcd_fb_driver);
}
staticvoid __exit lcd_exit(void)
{
    /*注销平台设备*/
    platform_driver_unregister(&lcd_fb_driver);
}
module_init(lcd_init);
module_exit(lcd_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Huang Gang");
MODULE_DESCRIPTION("My2440 LCD FrameBuffer Driver");
```

（2）实现 LCD 平台设备各接口函数，具体实现代码如下所示。

```
staticint __devinit lcd_fb_probe(struct   platform_device*pdev)
{
    int i;
    int ret;
    struct resource *res; /*用来保存从 LCD 平台设备中获取的 LCD 资源*/
    struct fb_info   *fbinfo;/*FrameBuffer 驱动所对应的 fb_info 结构体*/
    struct s3c2410fb_mach_info *mach_info; /*保存从内核中获取的平台设备数据*/
    struct my2440fb_var *fbvar; /*上面定义的驱动程序全局变量结构体*/
    /*LCD 屏的配置信息结构体，该结构体定义在 mach-s3c2410/include/mach/fb.h 中*/
    struct s3c2410fb_display *display;
    /*获取 LCD 硬件相关信息数据，在前面讲过内核使用 s3c24xx_fb_set_platdata 函数将 LCD 的硬件相关信息
保存到了 LCD 平台数据中，所以这里我们就从平台数据中取出来在驱动中使用*/
    mach_info = pdev->dev.platform_data;
    if(mach_info==NULL)
    {
        /*判断获取数据是否成功*/
        dev_err(&pdev->dev,"no platform   data for lcd/n");
        return-EINVAL;
    }
    /*获得在内核中定义的 FrameBuffer 平台设备的 LCD 配置信息结构体数据*/
    display = mach_info->displays+ mach_info->default_display;
     /*给 fb_info 分配空间，大小为 my2440fb_var 结构的内存，framebuffer_alloc 定义在 fb.h 中，在 fbsysfs.c
中实现*/
    fbinfo =   framebuffer_alloc(sizeof(struct   my2440fb_var),&pdev->dev);
    if(!fbinfo)
    {
        dev_err(&pdev->dev,"framebuffer alloc of registers   failed/n");
        ret =-ENOMEM;
        goto err_noirq;
    }
    /*重新将 LCD 平台设备数据设置为 fbinfo，以便在后面的一些函数中使用*/
    platform_set_drvdata(pdev, fbinfo);
    /*这里的用途其实就是将 fb_info 的成员 par（注意是一个 void 类型的指针）指向这里的私有变量结构体 fbvar，
目的是到其他接口函数中再取出 fb_info 的成员 par，从而能继续使用这里的私有变量*/
    fbvar = fbinfo->par;
     fbvar->dev=&pdev->dev;
```

```
    /*在系统定义的 LCD 平台设备资源中获取 LCD 中断号，platform_get_irq 定义在 platform_device.h 中*/
    fbvar->lcd_irq_no= platform_get_irq(pdev, 0);
    if(fbvar->lcd_irq_no< 0)
    {
        /*判断获取中断号是否成功*/
        dev_err(&pdev->dev,"no lcd irq   for platform/n");
        return-ENOENT;
    }
/*获取 LCD 平台设备所使用的 I/O 端口资源，注意这个 IORESOURCE_MEM 标志和 LCD 平台设备中定义的一致*/
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if(res ==NULL)
    {
        /*判断获取资源是否成功*/
        dev_err(&pdev->dev,"failed to   get memory region resource/n");
        return-ENOENT;
    }
    /*申请 LCD  I/O 端口所占用的 I/O 空间（注意理解 I/O 空间和内存空间的区别），request_mem_region 定义
在 ioport.h 中*/
    fbvar->lcd_mem= request_mem_region(res->start, res->end-res->start+ 1, pdev->name);
    if(fbvar->lcd_mem==NULL)
    {
        /*判断申请 I/O 空间是否成功*/
        dev_err(&pdev->dev,"failed to reserve memory   region/n");
        return-ENOENT;
    }
    /*将 LCD 的 I/O 端口占用的这段 I/O 空间映射到内存的虚拟地址，ioremap 定义在 io.h 中
      注意：I/O 空间要映射后才能使用，以后对虚拟地址的操作就是对 I/O 空间的操作*/
    fbvar->lcd_base= ioremap(res->start, res->end-res->start+ 1);
    if(fbvar->lcd_base==NULL)
    {
        /*判断映射虚拟地址是否成功*/
        dev_err(&pdev->dev,"ioremap() of registers   failed/n");
        ret =-EINVAL;
        goto err_nomem;
    }
    /*从平台时钟队列中获取 LCD 的时钟，这里为什么要取得这个时钟，从 LCD 屏的时序图上看，各种控制信号
的延迟都和 LCD 的时钟有关。系统的一些时钟定义在 arch/arm/plat-s3c24xx/s3c2410-clock.c 中*/
    fbvar->lcd_clock= clk_get(NULL,"lcd");
    if(!fbvar->lcd_clock)
    {
        /*判断获取时钟是否成功*/
        dev_err(&pdev->dev,"failed to   find lcd clock source/n");
        ret =-ENOENT;
        goto err_nomap;
    }
    /*时钟获取后要使能后才可以使用，clk_enable 定义在 arch/arm /plat-s3c/clock.c 中*/
    clk_enable(fbvar->lcd_clock);
    /*申请 LCD 中断服务，上面获取的中断号 lcd_fb_irq，使用快速中断方式:IRQF_DISABLED
      中断服务程序为:lcd_fb_irq，将 LCD 平台设备 pdev 作为参数传递过去了*/
    ret = request_irq(fbvar->lcd_irq_no, lcd_fb_irq, IRQF_DISABLED, pdev->name, fbvar);
    if(ret)
```

```
    {
        /*判断申请中断服务是否成功*/
        dev_err(&pdev->dev,"IRQ%d error    %d/n", fbvar->lcd_irq_no, ret);
        ret =-EBUSY;
        goto err_noclk;
    }
    /*好了，以上对要使用的资源进行了获取和设置。下面就开始初始化填充 fb_info 结构体*/
    /*首先初始化 fb_info 中代表 LCD 固定参数的结构体 fb_fix_screeninfo*/
    /*像素值与显示内存的映射关系有 5 种，定义在 fb.h 中。现在采用 FB_TYPE_PACKED_PIXELS 方式，在该
方式下，像素值与内存直接对应，例如在显示内存某单元写入一个“1”时，该单元对应的像素值也将是“1”，
这使得应用层把显示内存映射到用户空间变得非常方便。Linux 中当 LCD 为 TFT 屏时，显示驱动管理显示内存就
是基于这种方式*/
    strcpy(fbinfo->fix.id,driver_name);/*字符串形式的标识符*/
    fbinfo->fix.type= FB_TYPE_PACKED_PIXELS;
    /*以下这些根据 fb_fix_screeninfo 定义中的描述，当没有硬件时都设为 0*/
    fbinfo->fix.type_aux= 0;
    fbinfo->fix.xpanstep= 0;
    fbinfo->fix.ypanstep= 0;
    fbinfo->fix.ywrapstep= 0;
    fbinfo->fix.accel= FB_ACCEL_NONE;
    /*接着，再初始化 fb_info 中代表 LCD 可变参数的结构体 fb_var_screeninfo*/
    fbinfo->var.nonstd = 0;
    fbinfo->var.activate = FB_ACTIVATE_NOW;
    fbinfo->var.accel_flags = 0;
    fbinfo->var.vmode = FB_VMODE_NONINTERLACED;
    fbinfo->var.xres = display->xres;
    fbinfo->var.yres = display->yres;
    fbinfo->var.bits_per_pixel = display->bpp;
    /*指定对底层硬件操作的函数指针，因内容较多所以其定义在第（3）步中再讲*/
    fbinfo->fbops = &my2440fb_ops;
    fbinfo->flags = FBINFO_FLAG_DEFAULT;
    fbinfo->pseudo_palette = & fbvar->pseudo_pal;
    /*初始化色调色板(颜色表)为空*/
    for(i= 0; i< 256; i++)
    {
        fbvar->palette_buffer[i]= PALETTE_BUFF_CLEAR;
    }
    for(i= 0; i < mach_info->num_displays; i++)/*fb 缓存的长度*/
    {
        /*计算 FrameBuffer 缓存的最大大小，这里右移 3 位（即除以 8）是因为色位模式 BPP 是以位为单位*/
        unsignedlong smem_len=(mach_info->displays[i].xres* mach_info->displays[i].yres* mach_info->displays[i].
bpp)>> 3;
        if(fbinfo->fix.smem_len< smem_len)
        {
            fbinfo->fix.smem_len= smem_len;
        }
    }
    /*初始化 LCD 控制器之前要延迟一段时间*/
    msleep(1);
    /*初始化完 fb_info 后，开始对 LCD 各寄存器进行初始化，其定义在后面讲*/
    my2440fb_init_registers(fbinfo);
```

```
    /*初始化完寄存器后，开始检查 fb_info 中的可变参数，其定义在后面讲*/
    my2440fb_check_var(fbinfo);
    /*申请帧缓冲设备 fb_info 的显示缓冲区空间，其定义在后面讲*/
    ret = my2440fb_map_video_memory(fbinfo);
    if(ret)
    {
        dev_err(&pdev->dev,"failed to  allocate video RAM: %d/n", ret);
        ret =-ENOMEM;
        goto err_nofb;
    }
    /*最后，注册这个帧缓冲设备 fb_info 到系统中，register_framebuffer 定义在 fb.h 中，在 fbmem.c 中实现*/
    ret = register_framebuffer(fbinfo);
    if(ret < 0)
    {
        dev_err(&pdev->dev,"failed to register framebuffer  device: %d/n", ret);
        goto err_video_nomem;
    }
    /*对设备文件系统的支持（对设备文件系统的理解可参阅：嵌入式 Linux 之我行——设备文件系统剖析与使用）
创建 frambuffer 设备文件，device_create_file 定义在 linux/device.h 中*/
    ret = device_create_file(&pdev->dev,&dev_attr_debug);
    if(ret)
    {
        dev_err(&pdev->dev,"failed to  add debug attribute/n");
    }
    return 0;
/*以下是上面错误处理的跳转点*/
err_nomem:
    release_resource(fbvar->lcd_mem);
    kfree(fbvar->lcd_mem);
err_nomap:
    iounmap(fbvar->lcd_base);
err_noclk:
    clk_disable(fbvar->lcd_clock);
    clk_put(fbvar->lcd_clock);
err_noirq:
    free_irq(fbvar->lcd_irq_no, fbvar);
err_nofb:
    platform_set_drvdata(pdev,NULL);
    framebuffer_release(fbinfo);
err_video_nomem:
    my2440fb_unmap_video_memory(fbinfo);
    return ret;
}
/*LCD 中断服务程序*/
static irqreturn_t lcd_fb_irq(int irq,void*dev_id)
{
    struct my2440fb_var     *fbvar = dev_id;
    void __iomem *lcd_irq_base;
    unsignedlong lcdirq;
    /*LCD 中断挂起寄存器基地址*/
    lcd_irq_base = fbvar->lcd_base+ S3C2410_LCDINTBASE;
```

```
    /*读取 LCD 中断挂起寄存器的值*/
    lcdirq = readl(lcd_irq_base+ S3C24XX_LCDINTPND);
    /*判断是否为中断挂起状态*/
    if(lcdirq & S3C2410_LCDINT_FRSYNC)
    {
        /*填充调色板*/
        if(fbvar->palette_ready)
        {
            my2440fb_write_palette(fbvar);
        }
        /*设置帧已插入中断请求*/
        writel(S3C2410_LCDINT_FRSYNC, lcd_irq_base+ S3C24XX_LCDINTPND);
        writel(S3C2410_LCDINT_FRSYNC, lcd_irq_base+ S3C24XX_LCDSRCPND);
    }
    return IRQ_HANDLED;
}
/*填充调色板*/
staticvoid my2440fb_write_palette(struct   my2440fb_var*fbvar)
{
    unsignedint i;
    void __iomem *regs= fbvar->lcd_base;
    fbvar->palette_ready= 0;
    for(i = 0; i < 256; i++)
    {
        unsignedlong ent= fbvar->palette_buffer[i];
        if(ent ==   PALETTE_BUFF_CLEAR)
        {
            continue;
        }
        writel(ent, regs+   S3C2410_TFTPAL(i));
        if(readw(regs+ S3C2410_TFTPAL(i))== ent)
        {
            fbvar->palette_buffer[i]= PALETTE_BUFF_CLEAR;
        }
        else
        {
            fbvar->palette_ready= 1;
        }
    }
}
/*LCD 各寄存器进行初始化*/
staticint my2440fb_init_registers(struct fb_info*fbinfo)
{
    unsignedlong flags;
    void __iomem *tpal;
    void __iomem *lpcsel;
    /*从 lcd_fb_probe 探测函数设置的私有变量结构体中再获得 LCD 相关信息的数据*/
    struct my2440fb_var      *fbvar = fbinfo->par;
    struct s3c2410fb_mach_info *mach_info = fbvar->dev->platform_data;
    /*获得临时调色板寄存器基地址，S3C2410_TPAL 宏定义在 mach-s3c2410/include/mach/regs-lcd.h 中。注
意 lpcsel 是一个针对三星 TFT 屏的一个专用寄存器，如果用的不是三星的 TFT 屏可以不用管它*/
```

```
    tpal = fbvar->lcd_base+ S3C2410_TPAL;
    lpcsel = fbvar->lcd_base+ S3C2410_LPCSEL;
    /*在修改下面寄存器值之前先屏蔽中断，将中断状态保存到 flags 中*/
    local_irq_save(flags);
    /*这里就是在前面讲到的把 I/O 端口 C 和 D 配置成 LCD 模式*/
    modify_gpio(S3C2410_GPCUP, mach_info->gpcup, mach_info->gpcup_mask);
    modify_gpio(S3C2410_GPCCON, mach_info->gpccon, mach_info->gpccon_mask);
    modify_gpio(S3C2410_GPDUP, mach_info->gpdup, mach_info->gpdup_mask);
    modify_gpio(S3C2410_GPDCON, mach_info->gpdcon, mach_info->gpdcon_mask);
    /*恢复被屏蔽的中断*/
    local_irq_restore(flags);
    writel(0x00, tpal);/*临时调色板寄存器使能禁止*/
    /*在前面讲到过，它是三星 TFT 屏的一个寄存器，这里可以不管* /
    writel(mach_info->lpcsel, lpcsel);
    return 0;
}
/*该函数实现修改 GPIO 端口的值，注意第三个参数 mask 的作用是将要设置的寄存器值先清零*/
staticinlinevoid modify_gpio(void __iomem*reg,unsignedlongset,unsignedlong mask)
{
    unsignedlong tmp;
    tmp = readl(reg)&~mask;
    writel(tmp |set, reg);
}
/*检查 fb_info 中的可变参数*/
staticint my2440fb_check_var(struct fb_info*fbinfo)
{
    unsigned i;
    /*从 lcd_fb_probe 探测函数设置的平台数据中再获得 LCD 相关信息的数据*/
    struct fb_var_screeninfo *var =&fbinfo->var;/*fb_info 中的可变参数*/
    struct my2440fb_var *fbvar = fbinfo->par;/*在 lcd_fb_probe 探测函数中设置的私有结构体数据*/
    /*LCD 的配置结构体数据，这个配置结构体的赋值在前面的“2. 帧缓冲设备作为平台设备”中*/
    struct s3c2410fb_mach_info *mach_info = fbvar->dev->platform_data;
    struct s3c2410fb_display *display =NULL;
    struct s3c2410fb_display *default_display = mach_info->displays+ mach_info->default_display;
    /*LCD 的类型，看前面的“2. 帧缓冲设备作为平台设备”中的 type 赋值是 TFT 类型*/
    int type = default_display->type;
    /*验证 X/Y 解析度*/
    if(var->yres== default_display->yres&&
        var->xres== default_display->xres&&
        var->bits_per_pixel== default_display->bpp)
    {
        display = default_display;
    }
    else
    {
        for(i = 0; i < mach_info->num_displays; i++)
        {
            if(type== mach_info->displays[i].type&&
              var->yres== mach_info->displays[i].yres&&
              var->xres== mach_info->displays[i].xres&&
              var->bits_per_pixel== mach_info->displays[i].bpp)
```

```
            {
                display = mach_info->displays+ i;
                break;
            }
        }
    }
    if(!display)
    {
        return-EINVAL;
    }
    /*配置 LCD 配置寄存器 1 中的 5～6 位（配置成 TFT 类型）和配置 LCD 配置寄存器 5*/
    fbvar->regs.lcdcon1= display->type;
    fbvar->regs.lcdcon5= display->lcdcon5;
    /* 设置屏幕的虚拟解析像素和高度、宽度 */
    var->xres_virtual= display->xres;
    var->yres_virtual= display->yres;
    var->height= display->height;
    var->width = display->width;
    /* 设置时钟像素，行、帧切换值，水平同步、垂直同步长度值 */
    var->pixclock= display->pixclock;
    var->left_margin= display->left_margin;
    var->right_margin= display->right_margin;
    var->upper_margin= display->upper_margin;
    var->lower_margin= display->lower_margin;
    var->vsync_len= display->vsync_len;
    var->hsync_len= display->hsync_len;
    /*设置透明度*/
    var->transp.offset= 0;
    var->transp.length= 0;
    /*根据色位模式（BPP）设置可变参数中 R、G、B 的颜色位域。对于这些参数值的设置可参考 CPU 数据
    手册中“显示缓冲区与显示点对应关系图”*/
    switch(var->bits_per_pixel)
    {
        case 1:
        case 2:
        case 4:
            var->red.offset = 0;
            var->red.length = var->bits_per_pixel;
            var->green = var->red;
            var->blue = var->red;
            break;
        case 8:/* 8 bpp 332 */
            if(display->type!=   S3C2410_LCDCON1_TFT)
            {
                var->red.length = 3;
                var->red.offset = 5;
                var->green.length = 3;
                var->green.offset = 2;
                var->blue.length = 2;
                var->blue.offset = 0;
            }else{
```

```
                var->red.offset = 0;
                var->red.length = 8;
                var->green = var->red;
                var->blue = var->red;
            }
            break;
        case 12:/* 12 bpp 444 */
             var->red.length = 4;
            var->red.offset = 8;
            var->green.length = 4;
            var->green.offset = 4;
            var->blue.length = 4;
            var->blue.offset = 0;
            break;
        case 16:/* 16 bpp */
            if(display->lcdcon5& S3C2410_LCDCON5_FRM565)
            {
                /* 565 format */
                var->red.offset = 11;
                var->green.offset = 5;
                var->blue.offset = 0;
                var->red.length = 5;
                var->green.length = 6;
                var->blue.length = 5;
            }else{
                /* 5551 format */
                var->red.offset = 11;
                var->green.offset = 6;
                var->blue.offset = 1;
                var->red.length = 5;
                var->green.length = 5;
                var->blue.length = 5;
            }
            break;
        case 32:/* 24 bpp 888 and 8 dummy */
            var->red.length = 8;
            var->red.offset = 16;
            var->green.length = 8;
            var->green.offset = 8;
            var->blue.length = 8;
            var->blue.offset = 0;
            break;
    }
    return 0;
}
/*申请帧缓冲设备 fb_info 的显示缓冲区空间*/
staticint __init my2440fb_map_video_memory(struct fb_info*fbinfo)
{
    dma_addr_t map_dma;/*用于保存 DMA 缓冲区总线地址*/
    /*获得在 lcd_fb_probe 探测函数中设置的私有结构体数据* /
    struct my2440fb_var *fbvar = fbinfo->par;
    /*获得 FrameBuffer 缓存的大小，PAGE_ALIGN 定义在 mm.h 中*/
```

```
    unsigned map_size = PAGE_ALIGN(fbinfo->fix.smem_len);
    /*将分配的一个写合并 DMA 缓存区设置为 LCD 屏幕的虚拟地址（对于 DMA 请参考 DMA 相关知识）
    dma_alloc_writecombine 定义在 arch/arm/mm /dma-mapping.c 中*/
    fbinfo->screen_base= dma_alloc_writecombine(fbvar->dev, map_size,&map_dma, GFP_KERNEL);
    if(fbinfo->screen_base)
    {
        /*设置这片 DMA 缓存区的内容为空*/
        memset(fbinfo->screen_base, 0x00, map_size);
        /*将 DMA 缓冲区总线地址设成 fb_info 不可变参数中 framebuffer 缓存的开始位置*/
        fbinfo->fix.smem_start= map_dma;
    }
    return fbinfo->screen_base? 0 :-ENOMEM;
}
/*释放帧缓冲设备 fb_info 的显示缓冲区空间*/
staticinlinevoid my2440fb_unmap_video_memory(struct fb_info*fbinfo)
{
    struct my2440fb_var *fbvar = fbinfo->par;
    unsigned map_size = PAGE_ALIGN(fbinfo->fix.smem_len);
    /*与申请 DMA 的地方相对应*/
    dma_free_writecombine(fbvar->dev, map_size, fbinfo->screen_base, fbinfo->fix.smem_start);
}
/*LCD FrameBuffer 设备移除的实现，注意这里使用一个__devexit 宏，和 lcd_fb_probe 接口函数相对应。在 Linux
内核中，使用了大量不同的宏来标记具有不同作用的函数和数据结构，这些宏在 include/linux/init.h 头文件中定义，
编译器通过这些宏可以把代码优化放到合适的内存位置，以减少内存占用和提高内核效率。
  __devinit、__devexit 就是这些宏之一，在 probe()和 remove()函数中应该使用__devinit 和__devexit 宏。当
remove()函数使用了__devexit 宏时，则在驱动结构体中一定要使用__devexit_p 宏来引用 remove()，所以在第（1）
步中用__devexit_p 来引用 lcd_fb_remove 接口函数*/
staticint  __devexit lcd_fb_remove(struct platform_device*pdev)
{
    struct fb_info *fbinfo= platform_get_drvdata(pdev);
    struct my2440fb_var *fbvar = fbinfo->par;
    /*从系统中注销帧缓冲设备*/
    unregister_framebuffer(fbinfo);
    /*停止 LCD 控制器的工作*/
    my2440fb_lcd_enable(fbvar, 0);
    /*延迟一段时间，因为停止 LCD 控制器需要一点时间 */
    msleep(1);
    /*释放帧缓冲设备 fb_info 的显示缓冲区空间*/
    my2440fb_unmap_video_memory(fbinfo);
    /*将 LCD 平台数据清空和释放 fb_info 空间资源*/
    platform_set_drvdata(pdev,NULL);
    framebuffer_release(fbinfo);
    /*释放中断资源*/
    free_irq(fbvar->lcd_irq_no, fbvar);
    /*释放时钟资源*/
    if(fbvar->lcd_clock)
    {
        clk_disable(fbvar->lcd_clock);
        clk_put(fbvar->lcd_clock);
        fbvar->lcd_clock=NULL;
    }
```

```
    /*释放 LCD I/O 空间映射的虚拟内存空间*/
    iounmap(fbvar->lcd_base);
    /*释放申请的 LCD I/O 端口所占用的 I/O 空间*/
    release_resource(fbvar->lcd_mem);
    kfree(fbvar->lcd_mem);
    return 0;
}
/*停止 LCD 控制器的工作*/
staticvoid my2440fb_lcd_enable(struct my2440fb_var*fbvar,int enable)
{
    unsignedlong flags;
    /*在修改下面寄存器值之前先屏蔽中断，将中断状态保存到 flags 中*/
    local_irq_save(flags);
    if(enable)
    {
        fbvar->regs.lcdcon1|= S3C2410_LCDCON1_ENVID;
    }
    else
    {
        fbvar->regs.lcdcon1&=~S3C2410_LCDCON1_ENVID;
    }
    writel(fbvar->regs.lcdcon1,   fbvar->lcd_base+ S3C2410_LCDCON1);
    /*恢复被屏蔽的中断*/
    local_irq_restore(flags);
}
/*对 LCD FrameBuffer 平台设备驱动电源管理的支持，CONFIG_PM 这个宏定义在内核中*/
#ifdef CONFIG_PM
/*当配置内核时选上电源管理，则平台设备的驱动就支持挂起和恢复功能*/
staticint   lcd_fb_suspend(struct platform_device*pdev, pm_message_t state)
{
    /*挂起 LCD 设备，注意这里挂起 LCD 时并没有保存 LCD 控制器的各种状态，所以在恢复后 LCD 不会继续显
示挂起前的内容。若要继续显示挂起前的内容，则要在这里保存 LCD 控制器的各种状态*/
    struct fb_info *fbinfo= platform_get_drvdata(pdev);
    struct my2440fb_var *fbvar = fbinfo->par;
    /*停止 LCD 控制器的工作*/
    my2440fb_lcd_enable(fbvar, 0);
    msleep(1);
    /*停止时钟*/
    clk_disable(fbvar->lcd_clock);
    return 0;
}
static int lcd_fb_resume(struct   platform_device*pdev)
{
    /*恢复挂起的 LCD 设备*/
    struct fb_info *fbinfo= platform_get_drvdata(pdev);
    struct my2440fb_var *fbvar = fbinfo->par;
    /*开启时钟*/
    clk_enable(fbvar->lcd_clock);
    /*初始化 LCD 控制器之前要延迟一段时间*/
    msleep(1);
    /*恢复时重新初始化 LCD 各寄存器*/
```

```
    my2440fb_init_registers(fbinfo);
    /*重新激活 fb_info 中所有的参数配置，该函数定义在第（3）步中再讲*/
    my2440fb_activate_var(fbinfo);
    /*正与挂起时讲到的那样，因为没保存挂起时 LCD 控制器的各种状态，所以恢复后就让 LCD 显示空白，该函数定义也在第（3）步中再讲*/
    my2440fb_blank(FB_BLANK_UNBLANK, fbinfo);
    return 0;
}
#else
/*如果配置内核时没选上电源管理，则平台设备的驱动就不支持挂起和恢复功能，这两个函数也就无须实现了*/
#define lcd_fb_suspend    NULL
#define lcd_fb_resume    NULL
#endif
```

（3）帧缓冲设备驱动对底层硬件操作的函数接口的实现代码如下所示。

```
/*Framebuffer 底层硬件操作各接口函数*/
staticstruct fb_ops my2440fb_ops=
{
    .owner              = THIS_MODULE,
    .fb_check_var     = my2440fb_check_var,/*第（2）步中已实现*/
    .fb_set_par        = my2440fb_set_par,/*设置 fb_info 中的参数，主要是 LCD 的显示模式*/
    .fb_blank           = my2440fb_blank,/*显示空白（即 LCD 开关控制）*/
    .fb_setcolreg     = my2440fb_setcolreg,/*设置颜色表*/
    /*以下 3 个函数是可选的，主要是提供 fb_console 的支持，在内核中已经实现，这里直接调用即可*/
    .fb_fillrect       = cfb_fillrect,/*定义在 drivers/video/cfbfillrect.c 中*/
    .fb_copyarea       = cfb_copyarea,/*定义在 drivers/video/cfbcopyarea.c 中*/
    .fb_imageblit     = cfb_imageblit,/*定义在 drivers/video/cfbimgblt.c 中*/
};
/*设置 fb_info 中的参数，这里根据用户设置的可变参数 var 调整固定参数 fix*/
staticint my2440fb_set_par(struct fb_info*fbinfo)
{
    /*获得 fb_info 中的可变参数*/
    struct fb_var_screeninfo *var =&fbinfo->var;
    /*判断可变参数中的色位模式，根据色位模式来设置色彩模式* /
    switch(var->bits_per_pixel)
    {
        case 32:
        case 16:
        case 12:/*12BPP 时，设置为真彩色（分成红、绿、蓝三基色）*/
            fbinfo->fix.visual= FB_VISUAL_TRUECOLOR;
            break;
        case  1:/*1BPP  时，设置为黑白色（分黑、白两种色，FB_VISUAL_MONO01  代表黑，FB_VISUAL_MONO10 代表白）*/
            fbinfo->fix.visual= FB_VISUAL_MONO01;
            break;
        default:/*默认设置为伪彩色，采用索引颜色显示*/
            fbinfo->fix.visual= FB_VISUAL_PSEUDOCOLOR;
            break;
    }
    /*设置 fb_info 中固定参数中一行的字节数，公式：1 行字节数=(1 行像素个数×每像素位数 BPP)/8 */
     fbinfo->fix.line_length=(var->xres_virtual* var->bits_per_pixel)/ 8;
    /*修改以上参数后，重新激活 fb_info 中的参数配置（即使修改后的参数在硬件上生效）*/
```

```
    my2440fb_activate_var(fbinfo);
    return 0;
}
/*重新激活 fb_info 中的参数配置*/
staticvoid my2440fb_activate_var(struct   fb_info*fbinfo)
{
    /*获得结构体变量*/
    struct   my2440fb_var *fbvar = fbinfo->par;
    void __iomem *regs= fbvar->lcd_base;
    /*获得 fb_info 可变参数*/
    struct fb_var_screeninfo *var =&fbinfo->var;
    /*计算 LCD 控制寄存器 1 中的 CLKVAL 值，根据数据手册中该寄存器的描述，计算公式如下：
    * STN 屏：VCLK = HCLK / (CLKVAL * 2),   CLKVAL 要求>= 2
    * TFT 屏：VCLK = HCLK / [(CLKVAL + 1) * 2],   CLKVAL 要求>= 0*/
    int clkdiv =   my2440fb_calc_pixclk(fbvar, var->pixclock)/ 2;
    /*获得屏幕的类型*/
    int type = fbvar->regs.lcdcon1& S3C2410_LCDCON1_TFT;
    if(type == S3C2410_LCDCON1_TFT)
    {
        /*根据数据手册按照 TFT 屏的要求配置 LCD 控制寄存器 1～5*/
        my2440fb_config_tft_lcd_regs(fbinfo,&fbvar->regs);
        --clkdiv;
        if(clkdiv< 0)
        {
            clkdiv = 0;
        }
    }
    else
    {
        /*根据数据手册按照 STN 屏的要求配置 LCD 控制寄存器 1～5*/
        my2440fb_config_stn_lcd_regs(fbinfo,&fbvar->regs);
        if(clkdiv< 2)
        {
            clkdiv = 2;
        }
    }
    /*设置计算的 LCD 控制寄存器 1 中的 CLKVAL 值*/
    fbvar->regs.lcdcon1|=   S3C2410_LCDCON1_CLKVAL(clkdiv);
    /*将各参数值写入 LCD 控制寄存器 1～5 中*/
    writel(fbvar->regs.lcdcon1&~S3C2410_LCDCON1_ENVID, regs+ S3C2410_LCDCON1);
    writel(fbvar->regs.lcdcon2, regs+ S3C2410_LCDCON2);
    writel(fbvar->regs.lcdcon3, regs+ S3C2410_LCDCON3);
    writel(fbvar->regs.lcdcon4, regs+ S3C2410_LCDCON4);
    writel(fbvar->regs.lcdcon5, regs+ S3C2410_LCDCON5);
    /*配置帧缓冲起始地址寄存器 1～3*/
    my2440fb_set_lcdaddr(fbinfo);
    fbvar->regs.lcdcon1|= S3C2410_LCDCON1_ENVID,
    writel(fbvar->regs.lcdcon1, regs+ S3C2410_LCDCON1);
}
/*计算 LCD 控制寄存器 1 中的 CLKVAL 值*/
staticunsignedint my2440fb_calc_pixclk(struct my2440fb_var*fbvar,unsignedlong pixclk)
```

```
{
    /*获得 LCD 的时钟*/
    unsignedlong clk= clk_get_rate(fbvar->lcd_clock);
    /* 像素时钟单位是皮秒，而时钟的单位是赫兹，所以计算公式为：
     * Hz -> picoseconds is / 10^-12
     */
    unsignedlonglongdiv=(unsignedlonglong)clk* pixclk;
    div>>= 12;          /* div / 2^12 */
    do_div(div, 625* 625UL * 625);/* div / 5^12，do_div 宏定义在 asm/div64.h 中*/
    returndiv;
}
/*根据数据手册按照 TFT 屏的要求配置 LCD 控制寄存器 1～5*/
staticvoid my2440fb_config_tft_lcd_regs(conststruct fb_info*fbinfo,struct   s3c2410fb_hw*regs)
{
    conststruct my2440fb_var*fbvar = fbinfo->par;
    conststruct fb_var_screeninfo*var =&fbinfo->var;
    /*根据色位模式设置 LCD 控制寄存器 1 和 5，参考数据手册* /
    switch(var->bits_per_pixel)
    {
        case 1:/*1BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_TFT1BPP;
            break;
        case 2:/*2BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_TFT2BPP;
            break;
        case 4:/*4BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_TFT4BPP;
            break;
        case 8:/*8BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_TFT8BPP;
            regs->lcdcon5|= S3C2410_LCDCON5_BSWP| S3C2410_LCDCON5_FRM565;
            regs->lcdcon5&=~S3C2410_LCDCON5_HWSWP;
            break;
        case 16:/*16BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_TFT16BPP;
            regs->lcdcon5&=~S3C2410_LCDCON5_BSWP;
            regs->lcdcon5|= S3C2410_LCDCON5_HWSWP;
            break;
        case 32:/*32BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_TFT24BPP;
            regs->lcdcon5&=~(S3C2410_LCDCON5_BSWP|  S3C2410_LCDCON5_HWSWP  |  S3C2410_
LCDCON5_BPP24BL);
            break;
        default:/*无效的 BPP*/
            dev_err(fbvar->dev,"invalid bpp %d/n", var->bits_per_pixel);
    }
    /*设置 LCD 配置寄存器 2、3、4*/
    regs->lcdcon2=   S3C2410_LCDCON2_LINEVAL(var->yres-1)|
            S3C2410_LCDCON2_VBPD(var->upper_margin-1)|
            S3C2410_LCDCON2_VFPD(var->lower_margin-1)|
            S3C2410_LCDCON2_VSPW(var->vsync_len-1);
```

```
    regs->lcdcon3=   S3C2410_LCDCON3_HBPD(var->right_margin-1)|
                S3C2410_LCDCON3_HFPD(var->left_margin-1)|
                S3C2410_LCDCON3_HOZVAL(var->xres-1);
    regs->lcdcon4=   S3C2410_LCDCON4_HSPW(var->hsync_len-1);
}
/*根据数据手册按照 STN 屏的要求配置 LCD 控制寄存器 1～5*/
staticvoid my2440fb_config_stn_lcd_regs(conststruct fb_info*fbinfo,struct   s3c2410fb_hw*regs)
{
    conststruct my2440fb_var*fbvar = fbinfo->par;
    conststruct fb_var_screeninfo*var =&fbinfo->var;
    int type = regs->lcdcon1&~S3C2410_LCDCON1_TFT;
    int hs = var->xres>> 2;
    unsigned wdly =(var->left_margin>> 4)-1;
    unsigned wlh =(var->hsync_len>> 4)-1;
    if(type != S3C2410_LCDCON1_STN4)
    {
        hs >>= 1;
    }
    /*根据色位模式设置 LCD 控制寄存器 1，参考数据手册*/
    switch(var->bits_per_pixel)
    {
        case 1:/*1BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_STN1BPP;
            break;
        case 2:/*2BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_STN2GREY;
            break;
        case 4:/*4BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_STN4GREY;
            break;
        case 8:/*8BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_STN8BPP;
            hs *= 3;
            break;
        case 12:/*12BPP*/
            regs->lcdcon1|= S3C2410_LCDCON1_STN12BPP;
            hs *= 3;
            break;
        default:/*无效的 BPP*/
            dev_err(fbvar->dev,"invalid bpp   %d/n", var->bits_per_pixel);
    }
    /*设置 LCD 配置寄存器 2、3、4，参考数据手册*/
    if(wdly > 3) wdly = 3;
    if(wlh > 3) wlh = 3;
    regs->lcdcon2= S3C2410_LCDCON2_LINEVAL(var->yres-1);
    regs->lcdcon3= S3C2410_LCDCON3_WDLY(wdly)|
                S3C2410_LCDCON3_LINEBLANK(var->right_margin/ 8)|
                S3C2410_LCDCON3_HOZVAL(hs -1);
    regs->lcdcon4= S3C2410_LCDCON4_WLH(wlh);
}
/*配置帧缓冲起始地址寄存器 1～3，参考数据手册*/
```

```
staticvoid my2440fb_set_lcdaddr(struct   fb_info*fbinfo)
{
    unsignedlong saddr1, saddr2, saddr3;
    struct my2440fb_var *fbvar = fbinfo->par;
    void __iomem *regs= fbvar->lcd_base;
    saddr1 = fbinfo->fix.smem_start>> 1;
    saddr2 = fbinfo->fix.smem_start;
    saddr2 += fbinfo->fix.line_length* fbinfo->var.yres;
    saddr2 >>= 1;
    saddr3 = S3C2410_OFFSIZE(0)|S3C2410_PAGEWIDTH((fbinfo->fix.line_length/ 2)& 0x3ff);
    writel(saddr1, regs+ S3C2410_LCDSADDR1);
    writel(saddr2, regs+ S3C2410_LCDSADDR2);
    writel(saddr3, regs+ S3C2410_LCDSADDR3);
}
/*显示空白，blank mode 有 5 种模式，定义在 fb.h 中，是一个枚举*/
staticint   my2440fb_blank(int blank_mode,struct fb_info*fbinfo)
{
    struct my2440fb_var *fbvar = fbinfo->par;
    void __iomem *regs= fbvar->lcd_base;
    /*根据显示空白的模式来设置 LCD 是开启还是停止*/
    if(blank_mode== FB_BLANK_POWERDOWN)
    {
        my2440fb_lcd_enable(fbvar, 0);/*在第（2）步中定义*/
    }
    else
    {
        my2440fb_lcd_enable(fbvar, 1);/*在第（2）步中定义*/
    }
    /*根据显示空白的模式来控制临时调色板寄存器*/
    if(blank_mode== FB_BLANK_UNBLANK)
    {
        /*临时调色板寄存器无效*/
        writel(0x0, regs+ S3C2410_TPAL);
    }
    else
    {
        /*临时调色板寄存器有效*/
        writel(S3C2410_TPAL_EN, regs+ S3C2410_TPAL);
    }
    return 0;
}
/*设置颜色表*/
staticint   my2440fb_setcolreg(unsigned regno,unsigned red,unsigned green,unsigned blue,unsigned transp,
struct fb_info*fbinfo)
{
    unsignedint val;
    struct my2440fb_var *fbvar = fbinfo->par;
    void __iomem *regs= fbvar->lcd_base;
    switch(fbinfo->fix.visual)
    {
        case FB_VISUAL_TRUECOLOR:
```

```
            /*真彩色*/
            if(regno< 16)
            {
                u32 *pal = fbinfo->pseudo_palette;
                val = chan_to_field(red,&fbinfo->var.red);
                val |=   chan_to_field(green,&fbinfo->var.green);
                val |= chan_to_field(blue,&fbinfo->var.blue);
                pal[regno]= val;
            }
            break;
        case FB_VISUAL_PSEUDOCOLOR:
            /*伪彩色*/
            if(regno<   256)
            {
                val =(red>> 0)&   0xf800;
                val |=(green>> 5)& 0x07e0;
                val |=(blue>> 11)& 0x001f;
                writel(val, regs+   S3C2410_TFTPAL(regno));
                /*修改调色板*/
                schedule_palette_update(fbvar, regno,   val);
            }
            break;
        default:
            return 1;
    }
    return 0;
}
staticinlineunsignedint chan_to_field(unsignedint   chan,struct fb_bitfield*bf)
{
    chan &= 0xffff;
    chan >>= 16-bf->length;
    return chan << bf->offset;
}
/*修改调色板*/
staticvoid schedule_palette_update(struct my2440fb_var*fbvar,unsignedint regno,unsignedint val)
{
    unsignedlong flags;
    unsignedlong irqen;
    /*LCD 中断挂起寄存器基地址*/
    void __iomem *lcd_irq_base= fbvar->lcd_base+ S3C2410_LCDINTBASE;
    /*在修改中断寄存器值之前先屏蔽中断，将中断状态保存到 flags 中*/
    local_irq_save(flags);
     fbvar->palette_buffer[regno]= val;
    /*判断调色板是否准备就绪*/
    if(!fbvar->palette_ready)
    {
        fbvar->palette_ready= 1;
        /*使能中断屏蔽寄存器*/
        irqen = readl(lcd_irq_base+ S3C24XX_LCDINTMSK);
        irqen &=~S3C2410_LCDINT_FRSYNC;
        writel(irqen, lcd_irq_base+ S3C24XX_LCDINTMSK);
    }
```

```
	/*恢复被屏蔽的中断*/
	local_irq_restore(flags);
}
```

在上述代码中，第（1）部分代码的主要功能如下。

- ☑ 将 LCD 设备注册到系统平台设备中。
- ☑ 定义 LCD 平台设备结构体 lcd_fb_driver。

第（2）部分代码的主要功能如下。

- ☑ 获取和设置 LCD 平台设备的各种资源。
- ☑ 分配 fb_info 结构体空间。
- ☑ 初始化 fb_info 结构体中的各参数。
- ☑ 初始化 LCD 控制器。
- ☑ 检查 fb_info 中的可变参数。
- ☑ 申请帧缓冲设备的显示缓冲区空间。
- ☑ 注册 fb_info。

第（3）部分代码的主要功能如下。

- ☑ 实现对 fb_info 相关参数进行检查的硬件接口函数。
- ☑ 实现对 LCD 显示模式进行设定的硬件接口函数。
- ☑ 实现对 LCD 显示开关（空白）的硬件接口函数等。

18.9.2 编写访问 FrameBuffer 设备文件的驱动

下面将讲解编写一个向屏幕绘制矩形的驱动程序，此功能是通过/dev/graphics/fb0 设备文件实现的。实例文件 lcd.c 的具体实现代码如下所示。

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>
int main () {
	int fp=0;
	struct fb_var_screeninfo vinfo;
	struct fb_fix_screeninfo finfo;
	int screensize=0;
	char *fbp = 0;
	int x = 0, y = 0;
	int location = 0;
	int bytes_per_pixel;//组成每一个像素点的字节数
//以读写的方式打开/dev/graphics/fb0
	fp = open ("/dev/graphics/fb0",O_RDWR);

	if (fp < 0){
		printf("Error : Can not open framebuffer device\n");
		exit(1);
	}
//读取屏幕信息
	if (ioctl(fp,FBIOGET_FSCREENINFO,&finfo)){
```

```
            printf("Error reading fixed information\n");
            exit(2);
      }
      if (ioctl(fp,FBIOGET_VSCREENINFO,&vinfo)){
            printf("Error reading variable information\n");
            exit(3);
      }
      bytes_per_pixel = vinfo.bits_per_pixel / 8;
      //计算 FrameBuffer 存储空间的大小
      screensize = vinfo.xres * vinfo.yres * bytes_per_pixel;     //输出部分 LCD 信息
      printf("x=%d    y=%d    bytes_per_pixel=%d\n", vinfo.xres, vinfo.yres, bytes_per_pixel);
      printf("screensize=%d\n", screensize);
      //内存映射
      fbp =(char *) mmap (0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fp,0);
      if ((int) fbp == -1)
      {
             printf ("Error: failed to map framebuffer device to memory.\n");
             exit (4);
      }
      //使用双层循环矩形
      for(x=100;x<150;x++)
     {
            for(y=100;y<150;y++)
            {
                   location = x * bytes_per_pixel + y   *   finfo.line_length;

                   *(fbp + location) = 0;    /*  蓝色*/
                   *(fbp + location + 1) = 255; /*  绿色*/
                   *(fbp + location + 2) = 0; /*  红色*/
                   *(fbp + location + 3) = 0;    /*  是否透明*/
             }
      }
      munmap (fbp, screensize); /*解除映射*/
      close (fp);        /*关闭设备文件*/
      return 0;
}
```

执行脚本文件 build.sh，将手机连接到电脑后，会自动将编译好的 LCD 程序上传到手机中的/data/local 目录下。然后执行 LCD 程序会在手机屏幕中输出一个绿色的矩形。

18.9.3　在 S3C6410 下移植 FrameBuffer 驱动

将编写完成的 S3C_FB 代码复制到内核/drivers/video/目录下，FB 文件分别是 s3cfb.c、s3cfb_htk.c 和 s3cfb.h。其中，s3cfb.c 是主要的 FrameBuffer 驱动，s3cfb_htk.c 实现了 s3cfb.c 中的一些函数功能。在/drivers/video/中的 kconfig 中添加如下代码：

```
#add by chachi
config FB_S3C
tristate "S3C SMDK LCD framebuffer support"
depends on FB && ARCH_S3C64XX
select FB_CFB_FILLRECT
default n
```

```
---help---
TBA choice
depends on FB_S3C
prompt "Select LCD Type"
default FB_LTE480WV
config FB_LTD222OV
bool "LTD222OV"
---help---
TBA config FB_LTE246QV
bool "LTE246QV"
---help---
TBA config FB_LTS222QV
bool "LTS222QV"
---help---
TBA config FB_LTV350QV
bool "LTV350QV"
---help---
TBA config FB_LTE480WV
bool "LTE480WV/LTP700WV"
---help---
TBA config FB_LMS480QC
bool "LMS480QC0"
---help---
TBA config FB_HTKTECH
bool "HTKTECH FB"
---help---
TBA config FB_HTKTECH_800X480
bool "HTKTECH FB 640X480"
---help---
TBA config FB_HTKTECH_480X272
bool "HTKTECH FB 480X272"
---help---
TBA config FB_HTKTECH_A070VM04
bool "HTKTECH A070VM04 7inch 800x480"
select FB_CFB_COPYAREA
select FB_CFB_IMAGEBLIT
select FB_SOFT_CURSOR
---help---
TBA endchoice config FB_BPP
tristate "Advanced low level driver options"
depends on FB_S3C
default n
---help---
This enables tile blitting. Tile blitting is a drawing technique choice
depends on FB_BPP
prompt "Select BPP(Bit Per Pixel) type"
default FB_BPP_16
config FB_BPP_8
bool "8 BPP"
---help---
TBA config FB_BPP_16
```



```
bool "16 BPP"
---help---
TBA config FB_BPP_24
bool "24 BPP"
---help---
TBA config FB_BPP_32
bool "32 BPP"
---help---
TBA
endchoice choice
depends on FB_BPP
prompt "Choose postprocessing "
default PP_NOT_SUPPORTED
config PP_NOT_SUPPORTED
bool "Postprocessing NOT supported"
---help---
TBA config PP_S3C2443
bool "S3C2443 Postprocessing support"
depends on ARCH_S3C2443
---help---
TBA config PP_S3C2460
bool "S3C2460 Postprocessing support"
depends on ARCH_S3C2460
---help---
TBA
endchoice config FB_NUM
int "Number of S3C FB windows"
depends on ARCH_S3C64XX && FB_BPP
default "1"
---help---
TBA choice
depends on FB_BPP
prompt "Choose virtual screen support "
default VIRTUAL_SCREEN_NOT_SUPPORTED
config VIRTUAL_SCREEN_NOT_SUPPORTED
bool "Virtual screen NOT supported"
---help---
TBA config FB_VIRTUAL_SCREEN
bool "S3C virtual screen supported"
depends on ARCH_S3C64XX
---help---
TBA
endchoice choice
depends on FB_BPP
prompt "Choose double buffering support "
default DOUBLE_BUFFERING_NOT_SUPPORTED
config DOUBLE_BUFFERING_NOT_SUPPORTED
bool "double buffering NOT supported"
---help---
TBA config FB_DOUBLE_BUFFERING
bool "S3C double buffering supported"
```

```
depends on ARCH_S3C64XX
---help---
TBA
endchoice
#end change by chachi 在 makefile 文件中添加
#add by chachi
obj-$(CONFIG_FB_LTS222QV) += s3cfb.o s3c_lts222qv.o cfbimgblt.o cfbcopyarea.o cfbfillrect.o
obj-$(CONFIG_FB_LTD222OV) += s3cfb.o s3c_ltd222ov.o cfbimgblt.o cfbcopyarea.o cfbfillrect.o
obj-$(CONFIG_FB_LTV350QV) += s3cfb.o s3c_ltv350qv.o cfbimgblt.o cfbcopyarea.o cfbfillrect.o
obj-$(CONFIG_FB_LTE246QV) += s3cfb.o s3c_lte246qv.o cfbimgblt.o cfbcopyarea.o cfbfillrect.o
obj-$(CONFIG_FB_LTE480WV) += s3cfb.o s3c_lte480wv.o cfbimgblt.o cfbcopyarea.o cfbfillrect.o
obj-$(CONFIG_FB_LMS480QC) += s3cfb.o s3c_lms480qc.o cfbimgblt.o cfbcopyarea.o cfbfillrect.o
obj-$(CONFIG_FB_HTKTECH) += s3cfb.o s3c_htktech.o cfbimgblt.o cfbcopyarea.o cfbfillrect.o
obj-$(CONFIG_FB_HTKTECH_800X480) += s3cfb.o s3c_htktech.o cfbimgblt.o cfbcopyarea.o cfbfillrect.o
obj-$(CONFIG_FB_HTKTECH_480X272) += s3cfb.o s3c_htktech.o cfbimgblt.o cfbcopyarea.o cfbfillrect.o
obj-$(CONFIG_FB_HTKTECH_A070VM04) += s3cfb.o s3c_htktech.o cfbimgblt.o cfbcopyarea.o cfbfillrect.o
#end by chachi
```

然后在文件/arch/arm/mach-s3c6410/mach-smdk6410.c 中添加 platform_device，具体代码如下所示。

```
static struct resource s3c_lcd_resource[] = {
[0] = {
.start = S3C24XX_PA_LCD,
.end    = S3C24XX_PA_LCD + S3C24XX_SZ_LCD - 1,
.flags = IORESOURCE_MEM, [1] = {
.start = IRQ_LCD_VSYNC ,
.end    = IRQ_LCD_SYSTEM,
.flags = IORESOURCE_IRQ, }; static u64 s3c_device_lcd_dmamask = 0xffffffffUL; struct platform_device s3c_
device_lcd = {
.name = "s3c-lcd",
.id = -1,
.num_resources = ARRAY_SIZE(s3c_lcd_resource),
.resource = s3c_lcd_resource,
.dev = {
.dma_mask = &s3c_device_lcd_dmamask, //用来作为 framebuffer 驱动中申请 dma 缓存的掩码
.coherent_dma_mask = 0xffffffffUL };static struct platform_device *smdk6410_devices[]__initdata = { &s3c_
device_lcd, //+ chachi
};
```

然后在 smdk6410_iodesc 中添加下面的内容：

```
static struct map_desc smdk6410_iodesc[] = {
/* + chachi */
IODESC_ENT(LCD),
IODESC_ENT(HOSTIFB),
IODESC_ENT(SYSCON),
IODESC_ENT(GPIO),
/* end chachi */
};
```

其中下面的函数用于完成物理地址到虚拟地址的映射，6410 是带 MMU 的，内核中的地址基本上都是虚拟地址，一些物理地址在内核初始化时没有进行映射。笔者之前因为没有添加这几行，启动内核时总是会报 OOPS 错误。

```
#define IODESC_ENT(x) {
 (unsigned long)
```

```
S3C24XX_VA_##x,
__phys_to_pfn(S3C24XX_PA_##x),
 S3C24XX_SZ_##x,
 MT_DEVICE
 }
```

报错信息如下所示。

```
Unable to handle kernel paging request at virtual address XXX
```

在此可以自己设置一些用到的宏，其中 VA 表示虚拟地址，PA 表示物理地址。

```
#define S3C24XX_VA_LCD (0xF4500000) // ==S3C_VA_LCD at arch/plat-s3c/map.h
#define S3C24XX_PA_LCD S3C6400_PA_LCD // arch/arch-s3c2410/map.h
#define S3C6400_PA_LCD (0x77100000)
#define S3C24XX_SZ_LCD SZ_1M
#define S3C24XX_VA_HOSTIFB (0xF4C00000)
#define S3C24XX_PA_HOSTIFB (0x74100000)
#define S3C24XX_SZ_HOSTIFB SZ_1M
#define S3C24XX_VA_SYSCON (0xF6800000)
#define S3C24XX_PA_SYSCON (0x7E00F000)
#define S3C24XX_SZ_SYSCON SZ_4K
#define S3C24XX_VA_GPIO (0xF4600000)
#define S3C24XX_PA_GPIO (0x7F008000)
#define S3C24XX_SZ_GPIO SZ_4K 最后配置 menuconfig：
    Graphics support   --->
```

如果要修改 boot 时的 logo，可以在/drivers/video/logo/文件夹下修改，将复制过来的.c 文件名修改成 menuconfig 中选择的对应的.c 文件即可。将 Android 的 logo 复制过来，重命名一下，LCD 启动后就会显示一个机器人 logo。

第 19 章　音频系统驱动

在 Android 音频系统中，对应的硬件设备分为音频输入和音频输出两部分。手机终端中的输入设备通常是话筒，输出设备通常是耳机和扬声器。Android 音频系统的核心是 Audio 系统，此系统负责在音频方面的数据流传输和控制功能，也负责音频设备的管理功能。Audio 部分作为 Android 的 Audio 系统的输入/输出层次，一般负责播放 PCM 声音输出和从外部获取 PCM 声音，以及管理声音设备和设置。本章将详细讲解 Android 音频系统驱动的实现和移植内容，为读者学习本书后面的知识打下基础。

19.1　音频系统架构基础

在 Audio 系统中，整个音频管理模块主要分成以下 4 个层次。

（1）Media 库提供的 Audio 系统本地部分接口。

（2）AudioFlinger 作为 Audio 系统的中间层。

（3）Audio 的硬件抽象层提供底层支持。

（4）Audio 接口通过 JNI 和 Java 框架提供给上层。

Android 音频系统的基本层次结构如图 19-1 所示。

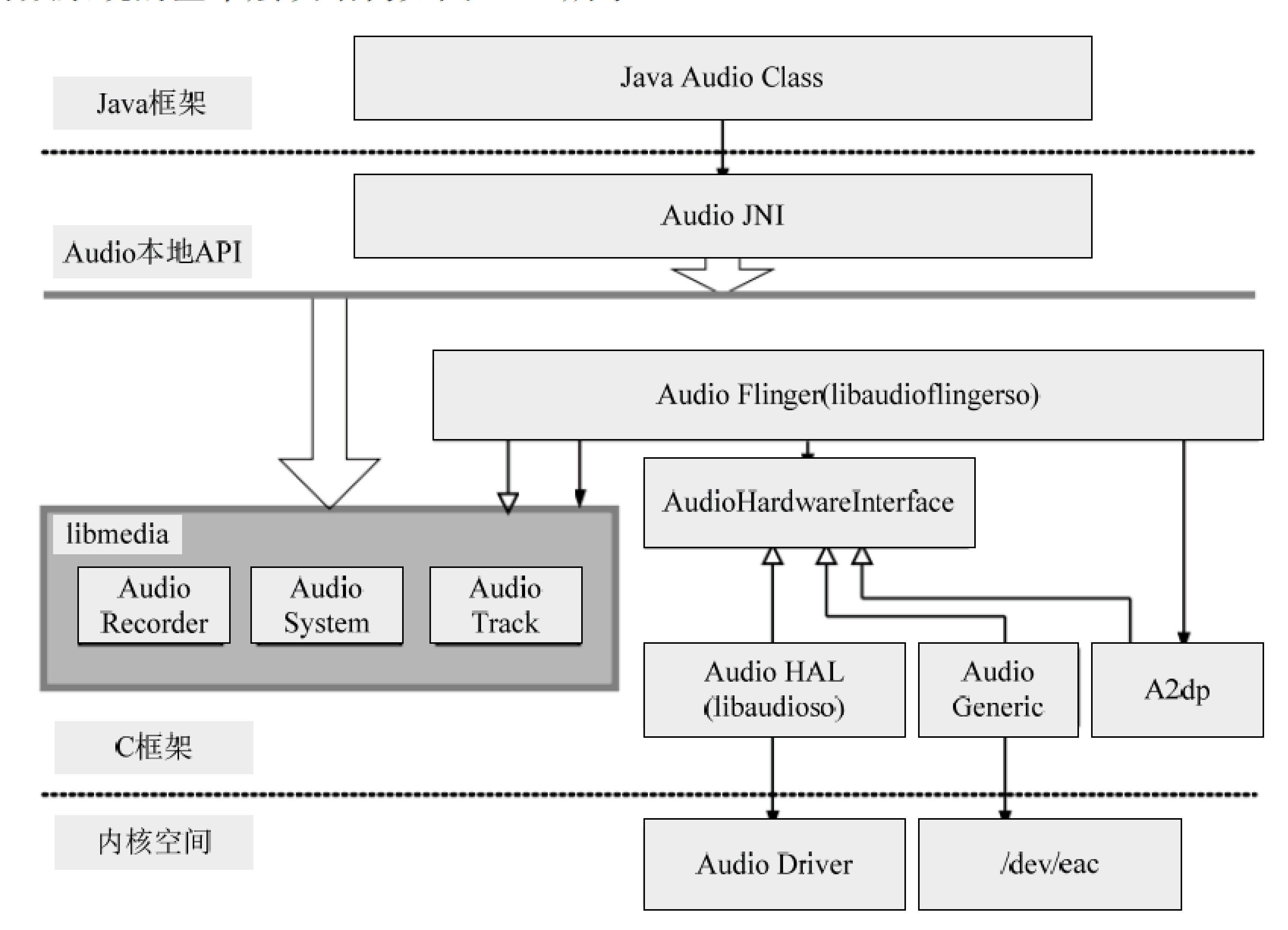


图 19-1　Android 音频系统的框架结构

图 19-1 中各个构成部分的具体说明如下。

（1）Audio 的 Java 部分

Java 部分的代码路径是 frameworks/base/media/java/android/media。

与 Audio 系统相关的 Java 包是 android.media，里面主要包含了与 AudioManager 和 Audio 系统等相关的类。

（2）Audio 的 JNI 部分

JNI 部分的代码路径是 frameworks/base/core/jni。

Audio 的 JNI 部分的生成库是 libandroid_runtime.so，Audio 的 JNI 是其中的一个部分。

（3）Audio 的框架部分

框架部分的头文件路径是 frameworks/base/include/media/。

具体实现源代码路径是 frameworks/base/media/libmedia/。

Audio 本地框架是 Media 库的一部分，本部分内容被编译成库 libmedia.so，提供 Audio 部分的接口（包括基于 Binder 的 IPC 机制）。

（4）Audio Flinger

Flinger 部分的代码路径是 frameworks/base/libs/audioflinger。

Flinger 部分的内容被编译成库 libaudioflinger.so，这是 Audio 系统的本地服务部分。

（5）Audio 的硬件抽象层接口

硬件抽象层接口的头文件路径是 hardware/libhardware_legacy/include/hardware/。

在各个系统中，Audio 硬件抽象层的具体实现可能是不同的，需要使用代码去继承相应的类并实现它们，作为 Android 系统本地框架层和驱动程序的接口。

19.1.1　层次说明

在 Audio 系统中，各个层次的具体说明如下所示。

（1）Audio 本地框架类：是 libmedia.so 的一个部分，这些 Audio 接口对上层提供接口，由下层的本地代码去实现。

（2）AudioFlinger：继承了 libmedia 中的接口，提供实现库 libaudioflinger.so。这部分内容没有自己的对外头文件，上层调用的只是 libmedia 本部分的接口，但实际调用的内容是 libaudioflinger.so。

（3）JNI：在 Audio 系统中，使用 JNI 和 Java 对上层提供接口，JNI 部分通过调用 libmedia 库提供的接口来实现。

（4）Audio 硬件抽象层：提供到硬件的接口，供 AudioFlinger 调用。Audio 的硬件抽象层实际上是各个平台开发过程中需要主要关注和独立完成的部分。

因为 Android 中的 Audio 系统不涉及编解码环节，只负责上层系统和底层 Audio 硬件的交互，所以通常以 PCM 作为输入/输出格式。

在 Android 的 Audio 系统中，无论上层还是下层，都使用一个管理类和“输出/输入”类来表示整个 Audio 系统，“输出/输入”类负责数据通道。Audio 系统在各个层次之间的对应关系如表 19-1 所示。

表 19-1　Android 各个层次的对应关系

层 次 说 明	Audio 管理环节	Audio 输出	Audio 输入
Java 层	android.media AudioSystem	android.media AudioTrack	android.media. AudioRecorder
本地框架层	AudioSystem	AudioTrack	AudioRecorder
AudioFlinger	IAudioFlinger	IAudioTrack	IAudioRecorder
硬件抽象层	AudioHardwareInterface	AudioStreamOut	AudioStreamIn

19.1.2　Media 库中的 Audio 框架

在 Media 库中提供了 Android 的 Audio 系统的核心框架，在库中实现了 AudioSystem、AudioTrack 和

AudioRecorder 3 个类。另外还提供了 IAudioFlinger 类接口，通过此类可以获得 IAudioTrack 和 IAudioRecorder 两个接口，分别用于声音的播放和录制功能。AudioTrack 和 AudioRecorder 分别通过调用 IAudioTrack 和 IAudioRecorder 来实现。

Audio 系统的头文件被保存在目录 frameworks/av/include/media 中。

其中包含的主要的头文件如下。

☑ AudioSystem.h：Media 库的 Audio 部分对上层的总管接口。
☑ IAudioFlinger.h：需要下层实现的总管接口。
☑ AudioTrack.h：放音部分对上接口。
☑ IAudioTrack.h：放音部分需要下层实现的接口。
☑ AudioRecorder.h：录音部分对上接口。
☑ IAudioRecorder.h：录音部分需要下层实现的接口。

其中文件 IAudioFlinger.h、IAudioTrack.h 和 IAudioRecorder.h 的接口是通过下层的继承来实现的。文件 AudioFlinger.h、AudioTrack.h 和 AudioRecorder.h 是对上层提供的接口，它们既供本地程序调用（例如声音的播放器、录制器等），也可以通过 JNI 向 Java 层提供接口。从具体功能上看，AudioSystem 用于综合管理 Audio 系统，而 AudioTrack 和 AudioRecorder 能够分别输出和输入音频数据，即分别实现播放和录制功能。

AudioTrack 是 Audio 输出环节的类，在里面包含了最重要的接口 write()，主要代码如下所示。

```
class AudioTrack : virtual public RefBase
{
public:
    enum channel_index {
        MONO     = 0,
        LEFT     = 0,
        RIGHT    = 1
    };
    enum event_type {
        EVENT_MORE_DATA = 0,           // Request to write more data to buffer.
                                       // If this event is delivered but the callback handler
                                       // does not want to write more data, the handler must explicitly
                                       // ignore the event by setting frameCount to zero.
        EVENT_UNDERRUN = 1,            // Buffer underrun occurred.
        EVENT_LOOP_END = 2,            // Sample loop end was reached; playback restarted from
                                       // loop start if loop count was not 0.
        EVENT_MARKER = 3,              // Playback head is at the specified marker position
                                       // (See setMarkerPosition()).
        EVENT_NEW_POS = 4,             // Playback head is at a new position
                                       // (See setPositionUpdatePeriod()).
        EVENT_BUFFER_END = 5           // Playback head is at the end of the buffer
    };
{
    typedef void (*callback_t)(int event,
void* user, void *info);
    AudioTrack( int streamType,
                uint32_t sampleRate = 0,     //音频的采样律
                int format = 0,              //音频的格式（例如 8 位或者 16 位的 PCM）
                int channelCount = 0,        //音频的通道数
                int frameCount = 0,          //音频的帧数
                uint32_t flags = 0,
```



```
                    callback_t cbf = 0,
                    void* user = 0,
                    int notificationFrames = 0);
    void        start();
    void        stop();
    void        flush();
    void        pause();
    void        mute(bool);
    ssize_t     write(const void* buffer, size_t size);
……………………………………………………………………
        enum {
            NO_MORE_BUFFERS = 0x80000001,   // same name in AudioFlinger.h, ok to be different value
            STOPPED = 1
        };
……………………………………………………………………
```

类 AudioRecord 是用于实现和 Audio 录制相关的功能，主要实现代码如下所示。

```
class AudioRecord
{
enum event_type {
    EVENT_MORE_DATA = 0,            // Request to read more data from PCM buffer.
    EVENT_OVERRUN = 1,              // PCM buffer overrun occurred.
    EVENT_MARKER = 2,               // Record head is at the specified marker position
                                    // (See setMarkerPosition()).
    EVENT_NEW_POS = 3,              // Record head is at a new position
                                    // (See setPositionUpdatePeriod()).
};
class Buffer
{
public:
    size_t frameCount;              // number of sample frames corresponding to size;
                                    // on input it is the number of frames available,
                                    // on output is the number of frames actually drained

    size_t size;                    // total size in bytes == frameCount * frameSize
    union {
        void* raw;
        short* i16;                 // signed 16-bit
        int8_t* i8;                 // unsigned 8-bit, offset by 0x80
    };
};
typedef void (*callback_t)(int event, void* user, void *info);
static status_t getMinFrameCount(size_t* frameCount,
                                  uint32_t sampleRate,
                                  audio_format_t format,
                                  audio_channel_mask_t channelMask);
```

在类 AudioTrack 和 AudioRecord 中，函数 read()和 write()的参数都是内存的指针及其大小，内存中的内容一般表示的是 Audio 的原始数据（PCM 数据）。这两个类还涉及 Audio 数据格式、通道数、帧数目等参数，可以在建立时指定，也可以在建立之后使用 set()函数进行设置。

另外，在 libmedia 库中提供的只是一个 Audio 系统框架，其中类 AudioSystem、AudioTrack 和 AudioRecord 分别调用下层的接口 IAudioFlinger、IAudioTrack 和 IAudioRecord 来实现。另外的一个接口是 IAudioFlingerClient，

它作为向 IAudioFlinger 中注册的监听器，相当于使用回调函数获取 IAudioFlinger 运行时的信息。

19.2　音频系统层次详解

在 Android 中，Audio 音频系统从上到下分别由 Java 的 Audio 类、Audio 本地框架类、AudioFlinger 和 Audio 的硬件抽象层几个部分组成，本节将简要介绍上述几个层次的基本知识。

19.2.1　本地代码详解

在 Android 系统中，AudioFlinger 是 Audio 音频系统的中间层，能够作为 libmedia 提供的 Audio 部分接口的实现。这部分本地代码的路径是 frameworks/base/libs/audioflinger。

文件 AudioFlinger.h 和 AudioFlinger.cpp 是实现 AudioFlinger 的核心文件，在里面提供了类 AudioFlinger，此类是一个 IAudioFlinger 的实现，其接口代码如下所示。

```
class AudioFlinger : public BnAudioFlinger,
public IBinder::DeathRecipient
{
public:
    static void instantiate();
    virtual status_t dump(int fd, const Vector<String16>& args);
    virtual sp<IAudioTrack> createTrack(
//获得音频输出接口（Track）
                                audio_stream_type_t streamType,
                                uint32_t sampleRate,
                                audio_format_t format,
                                audio_channel_mask_t channelMask,
                                size_t frameCount,
                                track_flags_t *flags,
                                const sp<IMemory>& sharedBuffer,
                                audio_io_handle_t output,
                                pid_t tid,   // -1 means unused, otherwise must be valid non-0
                                int *sessionId,
                                status_t *status) = 0;
//获得音频输出接口（Record）
    virtual sp<IAudioRecord> openRecord(
                                audio_io_handle_t input,
                                uint32_t sampleRate,
                                audio_format_t format,
                                audio_channel_mask_t channelMask,
                                size_t frameCount,
                                track_flags_t flags,
                                pid_t tid,   // -1 means unused, otherwise must be valid non-0
                                int *sessionId,
                                status_t *status) = 0;
```

由上述代码可以看出，AudioFlinger 使用函数 createTrack()来创建音频的输出设备 IAudioTrack，使用函数 openRecord()来创建音频的输入设备 IAudioRecord，并且还使用接口 get/set 来实现控制功能。

构造函数 AudioFlinger()的代码如下所示。



```
AudioFlinger::AudioFlinger()
{
    mHardwareStatus = AUDIO_HW_IDLE;
    mAudioHardware = AudioHardwareInterface::create();
    mHardwareStatus = AUDIO_HW_INIT;
    if (mAudioHardware->initCheck() == NO_ERROR) {
        mHardwareStatus = AUDIO_HW_OUTPUT_OPEN;
        status_t status;
        AudioStreamOut *hwOutput =
            mAudioHardware->openOutputStream (AudioSystem::PCM_16_BIT, 0, 0, &status);
        mHardwareStatus = AUDIO_HW_IDLE;
        if (hwOutput) {
            mHardwareMixerThread =
            new MixerThread(this, hwOutput, AudioSystem::AUDIO_OUTPUT_HARDWARE);
        } else {
            LOGE("Failed to initialize hardware    output stream, status: %d", status);
        }
    #ifdef WITH_A2DP
        mA2dpAudioInterface = new A2dpAudioInterface();
        AudioStreamOut  *a2dpOutput  =  mA2dpAudioInterface->openOutputStream(AudioSystem::PCM_16_
BIT, 0, 0, &status);
        if (a2dpOutput) {
            mA2dpMixerThread = new MixerThread(this, a2dpOutput, AudioSystem::AUDIO_OUTPUT_A2DP);
            if (hwOutput) {
                uint32_t  frameCount  =  ((a2dpOutput->bufferSize()/a2dpOutput->frameSize())  *  hwOutput->
sampleRate()) / a2dpOutput->sampleRate();
                MixerThread::OutputTrack *a2dpOutTrack = new MixerThread::OutputTrack(mA2dpMixerThread,
                hwOutput->sampleRate(),
                AudioSystem::PCM_16_BIT,
                hwOutput->channelCount(),
                frameCount);
                mHardwareMixerThread->setOuputTrack(a2dpOutTrack);
            }
        } else {
            LOGE("Failed to initialize A2DP output stream, status: %d", status);
        }
#endif
        setRouting(AudioSystem::MODE_NORMAL,  AudioSystem::ROUTE_SPEAKER,  AudioSystem::
ROUTE_ALL);
        setRouting(AudioSystem::MODE_RINGTONE,  AudioSystem::ROUTE_SPEAKER,  AudioSystem::
ROUTE_ALL);
        setRouting(AudioSystem::MODE_IN_CALL,  AudioSystem::ROUTE_EARPIECE,  AudioSystem::
ROUTE_ALL);
        setMode(AudioSystem::MODE_NORMAL);
        setMasterVolume(1.0f);
        setMasterMute(false);
        mAudioRecordThread = new    AudioRecordThread(mAudioHardware, this);
        if (mAudioRecordThread != 0) {
            mAudioRecordThread->run("AudioRecordThread", PRIORITY_URGENT_AUDIO);
        }
     } else {
```

```
        LOGE("Couldn't even initialize the stubbed audio hardware!");
    }
}
```

由上述代码可以看出，在初始化 AudioFlinger 之后，会首先获得放音设备，然后为混音器（Mixer）建立线程并建立放音设备线程，最后在线程中获得放音设备。

在文件 frameworks/av/services/audioflinger/AudioResampler.h 中定义了类 AudioResampler，此类是一个音频重取样器的工具类，定义代码如下所示。

```
class AudioResampler {
public:
    enum src_quality {
        DEFAULT=0,
        LOW_QUALITY=1,                //线性差值算法
        MED_QUALITY=2,                //立方差值算法
        HIGH_QUALITY=3                //fixed multi-tap FIR 算法
        VERY_HIGH_QUALITY=4,
    };
    static AudioResampler* create(int bitDepth, int inChannelCount,
            int32_t sampleRate, src_quality quality=DEFAULT_QUALITY);
    virtual ~AudioResampler();
    virtual void init() = 0;
    virtual void setSampleRate(int32_t inSampleRate);
    virtual void setVolume(int16_t left, int16_t right);
    virtual void setLocalTimeFreq(uint64_t freq);

    // set the PTS of the next buffer output by the resampler
    virtual void setPTS(int64_t pts);

    virtual void resample(int32_t* out, size_t outFrameCount,
            AudioBufferProvider* provider) = 0;

    virtual void reset();
    virtual size_t getUnreleasedFrames() const { return mInputIndex; }

    // called from destructor, so must not be virtual
    src_quality getQuality() const { return mQuality; }
```

在上述音频重取样工具类中，包含了如下 4 种质量。

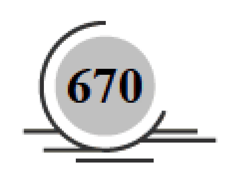
☑ 低等质量（LOW_QUALITY）：使用线性差值算法实现。
☑ 中等质量（MED_QUALITY）：使用立方差值算法实现。
☑ 高等质量（HIGH_ QUALITY）：使用 FIR（有限阶滤波器）实现。
☑ VERY_HIGH_QUALITY：非常高质量。

在类 AudioResampler 中，AudioResamplerOrder1 是线性实现，AudioResamplerCubic.*文件提供立方实现方式，AudioResamplerSinc.*提供 FIR 实现。

通过文件 AudioMixer.h 和 AudioMixer.cpp 实现了一个 Audio 系统混音器，它被 AudioFlinger 调用，一般用于声音输出之前的处理，提供多通道处理、声音缩放、重取样。AudioMixer 调用了 AudioResampler。

19.2.2 JNI 代码详解

在 Android 中的 Audio 系统中，通过 JNI 向 Java 层提供功能强大的接口，这样就可以在 Java 层通过 JNI



接口完成 Audio 系统的大部分操作。

Audio JNI 的实现代码保存在 frameworks/base/core/jni 目录下，在目录中主要有 3 个核心文件，这 3 个文件分别对应了 Android Java 框架中的 3 个类的支持，具体说明如下。

- ☑ android.media.AudioSystem：负责 Audio 系统的总体控制。
- ☑ android.media.AudioTrack：负责 Audio 系统的输出环节。
- ☑ android.media.AudioRecorder：负责 Audio 系统的输入环节。

在 Android 系统的 Java 层中，可以对 Audio 系统进行控制和数据流操作，其中控制操作和底层的处理基本一致；但是对于数据流操作，由于 Java 不支持指针，因此接口被封装成了另外的形式。例如在音频输出功能中，通过文件 android_media_AudioTrack.cpp 提供了写字节和写短整型的接口类型。对应代码如下所示。

```
static jint android_media_AudioTrack_native_
write(JNIEnv *env,   jobject thiz,
jbyteArray javaAudioData,
jint offsetInBytes, jint sizeInBytes,
jint javaAudioFormat) {
    jbyte* cAudioData = NULL;
    AudioTrack *lpTrack = NULL;
    lpTrack = (AudioTrack *)env->GetIntField(
                thiz, javaAudioTrackFields. Native TrackInJavaObj);
    ssize_t written = 0;
    if (lpTrack->sharedBuffer() == 0) {
    //进行写操作
        written = lpTrack->write(cAudioData +
offsetInBytes, sizeInBytes);
    } else {
        if (javaAudioFormat == javaAudioTrackFields.PCM16) {
            memcpy(lpTrack->sharedBuffer()->pointer(),
                    cAudioData+offsetInBytes, sizeInBytes);
            written = sizeInBytes;
        } else if (javaAudioFormat == javaAudioTrackFields.PCM8) {
            int count = sizeInBytes;
            int16_t *dst = (int16_t *)lpTrack->sharedBuffer()->pointer();
            const int8_t *src = (const int8_t *)
(cAudioData + offsetInBytes);
            while(count--) {
                *dst++ = (int16_t)(*src++^0x80) << 8;
            }
            written = sizeInBytes;
        }
    }
    env->ReleasePrimitiveArrayCritical(javaAudioData, cAudioData, 0);
    return (int)written;
}
```

19.2.3　Java 层代码详解

在 Android 的 Audio 系统中，和 Java 相关的类定义在包 android.media 中，Java 部分的代码保存在 frameworks/base/media/java/android/media 目录中，主要实现了如下所示的类。

- ☑ android.media.AudioSystem

- ☑ android.media. Audio Track
- ☑ android.media.AudioRecorder
- ☑ android.media.AudioFormat

其中前 3 个类和本地代码是对应的，在 AudioFormat 中提供了一些和 Audio 相关的枚举值。在此需要注意的是在 Audio 系统的 Java 代码中，虽然可以通过 AudioTrack 和 AudioRecorder 的 write()和 read()接口在 Java 层对 Audio 的数据流进行操作，但是更多的时候并不需要这样做，而是在本地代码中直接调用接口进行数据流的“输入/输出”，而在 Java 层只进行控制类方面的操作，不处理具体的数据流工作。

19.3 移植 Audio 系统

在 Android 系统中，Audio 的标准化部分是硬件抽象层的接口，所以需要针对不同的特定平台移植 Audio 驱动程序和 Audio 硬件抽象层，程序员的任务就是移植这两方面的内容，本节将详细讲解移植 Audio 系统所需要做的工作。

19.3.1 移植需要做的工作

在移植 Android 的 Audio 系统之前，需要先弄清如下两点。

（1）Audio 驱动程序

Audio 驱动程序部分需要在 Linux 内核中实现，并且大多数 Audio 驱动程序都需要提供用于音量控制等功能的控制类接口，通过这些接口实现 PCM 输入、输出的数据类接口。

（2）Audio 硬件抽象层

Audio 硬件抽象层是 Audio 驱动程序和 Audio 本地框架类 AudioFlinger 的接口。根据 Android 系统对接口的定义，Audio 硬件抽象层是 C++类的接口，需要在继承接口中定义如下 3 个类来实现 Audio 硬件抽象层。

- ☑ 实现总控类。
- ☑ 输入类。
- ☑ 输出类。

要想实现一个 Android 的硬件抽象层，则需要实现 AudioHardwareInterface、AudioStreamOut 和 AudioStreamIn 3 个类，并将代码编译成动态库 libaudio.so。AudioFlinger 会连接这个动态库，并调用其中的 createAudioHardware()函数来获取接口。

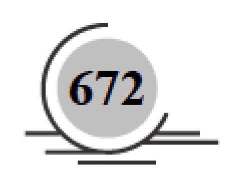
在文件 AudioHardwareBase.h 中定义了类 AudioHardwareBase，此类继承了 Audio HardwareInterface，通过继承此接口也可以实现 Audio 的硬件抽象层。Android 系统的 Audio 硬件抽象层可以通过继承类 AudioHardwareInterface 来实现，其中分为控制部分和“输入/输出”处理部分。

19.3.2 硬件抽象层移植分析

Audio 系统的硬件抽象层是 AudioFlinger 和 Audio 硬件之间的接口，在不同系统的移植过程中可以有不同的实现方式。其中 Audio 硬件抽象层的接口路径是 hardware/libhardware_legacy/include/hardware/。

在上述路径的核心文件是 AudioHardwareBase.h 和 AudioHardwareInterface.h。

作为 Android 系统的 Audio 硬件抽象层，既可以基于 Linux 标准的 ALSA 或 OSS 音频驱动来实现，也可以基于私有的 Audio 驱动接口来实现。



在文件 AudioHardwareInterface.h 中分别定义了类 AudioStreamOut、AudioStreamIn 和 AudioHardwareInterface。类 AudioStreamOut 和 AudioStreamIn 分别描述了音频输出设备和音频输入设备，其中负责数据流的接口分别是函数 wirte()和 read()，其参数是表示一块内存的指针和长度；另外还有一些设置和获取接口。类 AudioStreamOut 和 AudioStreamIn 的实现代码如下所示。

```
class AudioStreamOut {
public:
    virtual                 ~AudioStreamOut() = 0;
    virtual status_t      setVolume(float volume) = 0;
    virtual ssize_t       write(const void* buffer, size_t bytes) = 0;
    virtual int           channelCount() const = 0;
virtual int           format() const = 0;
virtual status_t      setVolume(float volume) = 0;
virtual ssize_t       write(const void* buffer, size_t bytes) = 0;
virtual status_t dump(int fd, const Vector<String16>& args) = 0;
};
class AudioStreamIn {
public:
    virtual                 ~AudioStreamIn() = 0;
    virtual status_t      setGain(float gain) = 0;
    virtual ssize_t       read(void* buffer, ssize_t bytes) = 0;
    virtual int           channelCount() const = 0;
    virtual int           format() const = 0;
    virtual status_t      setGain(float gain) = 0;
    virtual ssize_t       read(void* buffer, ssize_t bytes) = 0;
    virtual status_t dump(int fd, const Vector<String16>& args) = 0;
};
```

由此可见，类 AudioStreamOut 和 AudioStreamIn 是两户对应的接口类，分别实现输出和输入环节。类 AudioStreamOut 和 AudioStreamIn 都需要通过 Audio 硬件抽象层的核心 AudioHardwareInterface 接口类来获取。接口类 AudioHardwareInterface 的实现代码如下所示。

```
class AudioHardwareInterface {
public:
                            AudioHardwareInterface();
    virtual                 ~AudioHardwareInterface() { }
    virtual status_t      initCheck() = 0;
    virtual status_t      standby() = 0;
    virtual status_t      setVoiceVolume(float volume) = 0;
    virtual status_t      setMasterVolume(float volume) = 0;
    virtual status_t      setRouting(int mode, uint32_t routes);
    virtual status_t      getRouting(int mode, uint32_t* routes);
    virtual status_t      setMode(int mode);
    virtual status_t      getMode(int* mode);
    virtual status_t      setMicMute(bool state) = 0;
    virtual status_t      getMicMute(bool* state) = 0;
    virtual status_t      setParameter(const char* key, const char* value);
    virtual AudioStreamOut* openOutputStream(                    //打开输出流
                                    int format=0,
                                    int channelCount=0,
                                    uint32_t sampleRate=0) = 0;
    virtual AudioStreamIn* openInputStream(                      //打开输入流
```

```
                                   int format,
                                   int channelCount,
                                   uint32_t sampleRate) = 0;
    virtual status_t dumpState(int fd, const Vector<String16>& args);
    static AudioHardwareInterface* create();
```

在上述 AudioHardwareInterface 接口的实现代码中，分别使用函数 openOutputStream()和 openInputStream()来获取类 AudioStreamOut 和类 AudioStreamIn，将它们分别作为音频输入设备和输出设备来使用。

除此之外，在文件 AudioHardwareInterface.h 中还定义了 C 语言的接口来获取一个 AudioHardware Interface 类型的指针，具体的定义代码如下所示。

```
extern "C" AudioHardwareInterface* createAudioHardware(void);
```

19.3.3 AudioFlinger 中的 Audio 硬件抽象层

在 Android 系统的 AudioFlinger 中，可以通过编译宏的方式来选择到底用哪一个 Audio 硬件抽象层。可选择的 Audio 硬件抽象层既可以作为参考设计，也可以在没有实际的 Audio 硬件抽象层使用，目的是保证系统的正常运行。

1．编译文件

文件 Android.mk 是 AudioFlinger 的编译文件，定义代码如下所示。

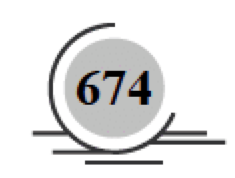
```
ifeq ($(strip $(BOARD_USES_GENERIC_AUDIO)),true)
  LOCAL_STATIC_LIBRARIES += libaudiointerface
else
  LOCAL_SHARED_LIBRARIES += libaudio
endif
LOCAL_MODULE:= libaudioflinger
include $(BUILD_SHARED_LIBRARY)
```

在上述代码中，当 BOARD_USES_GENERIC_AUDIO 为 True 时连接 libaudiointer face.a 静态库；当 BOARD_USES_GENERIC_AUDIO 为 False 时连接 libaudiointerface.so 动态库，在多数情况下使用后者。

另外，在文件 Android.mk 中也生成了 libaudiointerface.a，具体代码如下所示。

```
include $(CLEAR_VARS)
LOCAL_SRC_FILES:= \
    AudioHardwareGeneric.cpp \
    AudioHardwareStub.cpp \
    AudioDumpInterface.cpp \
    AudioHardwareInterface.cpp
LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    libmedia \
    libhardware_legacy
ifeq ($(strip $(BOARD_USES_GENERIC_AUDIO)),true)
  LOCAL_CFLAGS += -DGENERIC_AUDIO
endif
LOCAL_MODULE:= libaudiointerface
include $(BUILD_STATIC_LIBRARY)
```

在上述代码中，分别编译 4 个源文件来生成 libaudiointerface.a 静态库。其中文件 AudioHardwareInterface.cpp 用于实现基础类和管理；文件 AudioHardwareGeneric.cpp、AudioHard wareStub.cpp 和 AudioDumpInterface.cpp



分别代表一种 Audio 硬件抽象层的实现，具体说明如下。

☑　AudioHardwareGeneric.cpp：实现基于特定驱动的通用 Audio 硬件抽象层。

☑　AudioHardwareStub.cpp：实现 Audio 硬件抽象层的一个桩。

☑　AudioDumpInterface.cpp：实现输出到文件的 Audio 硬件抽象层。

在文件 AudioHardwareInterface.cpp 中定义了 AudioHardwareInterface::create()函数，此函数是 Audio 硬件抽象层的创建函数，主要实现代码如下所示。

```
AudioHardwareInterface* AudioHardwareInterface::create()
{
    AudioHardwareInterface* hw = 0;
    char value[PROPERTY_VALUE_MAX];
#ifdef GENERIC_AUDIO
    hw = new AudioHardwareGeneric();
//此处用通用的 Audio 硬件抽象层
#else
    if (property_get("ro.kernel.qemu", value, 0)) {
        LOGD("Running in emulation - using generic audio driver");
        hw = new AudioHardwareGeneric();
    }
    else {
        LOGV("Creating Vendor Specific AudioHardware");
        hw = createAudioHardware();
//此处用实际的 Audio 硬件抽象层
    }
#endif
    if (hw->initCheck() != NO_ERROR) {
        LOGW("Using stubbed audio hardware.No sound will be produced.");
        delete hw;
        hw = new AudioHardwareStub();
//此处用实际的 Audio 硬件抽象层的桩实现
    }
#ifdef DUMP_FLINGER_OUT
    hw = new AudioDumpInterface(hw);
//此处用实际的 Audio 的 Dump 接口实现
#endif
    return hw;
}
```

2. 桩方式实现

在文件 AudioHardwareStub.h 和 AudioHardwareStub.cpp 中，通过桩方式实现了一个 Android 硬件抽象层。桩方式不操作实际的硬件和文件，只是进行了空操作处理。当在系统中没有实际的 Audio 设备时才使用桩方式实现，这样可以保证系统的正常工作。如果使用这个硬件抽象层，实际上 Audio 系统的输入和输出都将为空。

在文件 AudioHardwareStub.h 中定义了类 AudioStreamOutStub 和类 AudioStreamInStub，功能是分别实现输入和输出，主要实现代码如下所示。

```
class AudioStreamOutStub : public AudioStreamOut {
public:
    virtual status_t     set(int format, int
channelCount, uint32_t sampleRate);
```

```
    virtual uint32_t      sampleRate() const { return 44100; }
    virtual size_t        bufferSize() const { return 4096; }
    virtual int           channelCount() const { return 2; }
    virtual int           format() const { return
 AudioSystem::PCM_16_BIT; }
    virtual uint32_t      latency() const { return 0; }
    virtual status_t      setVolume(float volume) { return NO_ERROR; }
    virtual ssize_t       write(const void* buffer, size_t bytes);
    virtual status_t      standby();
    virtual status_t      dump(int fd, const Vector<String16>& args);
};
class AudioStreamInStub : public AudioStreamIn {
public:
    virtual status_t set(int format, int
 channelCount, uint32_t sampleRate, AudioSystem::
audio_in_acoustics acoustics);
    virtual uint32_t      sampleRate() const { return 8000; }
    virtual size_t        bufferSize() const { return 320; }
    virtual int           channelCount() const { return 1; }
    virtual int           format() const { return
 AudioSystem::PCM_16_BIT; }
    virtual status_t      setGain(float gain) { return NO_ERROR; }
    virtual ssize_t       read(void* buffer, ssize_t bytes);
    virtual status_t      dump(int fd, const Vector<String16>& args);
    virtual status_t      standby() { return NO_ERROR; }
};
```

在上述代码中，只用缓冲区大小、采样率和通道数这 3 个固定的参数将一些函数直接无错误返回。

然后需要使用类 AudioHardwareStub 来继承类 AudioHardwareBase，也就是继承类 AudioHardwareInterface，主要实现代码如下所示。

```
  class AudioHardwareStub : public   AudioHardwareBase
  {
  public:
                                AudioHardwareStub();
        virtual                ~AudioHardwareStub();
        virtual status_t      initCheck();
        virtual status_t      setVoiceVolume(float volume);
        virtual status_t      setMasterVolume(float volume);
        virtual status_t      setMicMute(bool state)
{ mMicMute = state;   return   NO_ERROR; }
        virtual status_t      getMicMute(bool* state)
{ *state = mMicMute ; return NO_ERROR; }
        virtual status_t      setParameter(const
char* key, const char* value)
                 { return NO_ERROR; }
        virtual AudioStreamOut* openOutputStream(               //打开输出流
                                  int format=0,
                                  int channelCount=0,
                                  uint32_t sampleRate=0,
                                  status_t *status=0);
```

```
        virtual AudioStreamIn* openInputStream(              //打开输入流
                                int format,
                                int channelCount,
                                uint32_t sampleRate,
                                status_t *status,
                                AudioSystem::audio_in_acoustics acoustics);
  ...
```

为了保证可以输入和输出声音，桩实现的主要内容是实现类 AudioStreamOutStub 和类 AudioStreamInStub 的“读/写”函数，主要实现代码如下所示。

```
ssize_t AudioStreamOutStub::write(const void* buffer, size_t bytes)
{
    usleep(bytes * 1000000 / sizeof(int16_t) /
channelCount() / sampleRate());
    return bytes;
}
ssize_t AudioStreamInStub::read(void* buffer, ssize_t bytes)
{
    usleep(bytes * 1000000 / sizeof(int16_t) /
channelCount() / sampleRate());
    memset(buffer, 0, bytes);
    return bytes;
}
```

当使用这个接口来输入和输出音频时，和真实的设备并没有任何关系，输出和输入都使用延时来完成。在输出时不会播出声音，但是返回值表示全部内容已经输出完成；在输入时会返回全部为 0 的数据。

3. 通用 Audio 硬件抽象层

在 Android 系统中，文件 AudioHardwareGeneric.h 和 AudioHardwareGeneric.cpp 实现了通用的 Audio 硬件抽象层。与前面介绍的桩实现方式不同，这是一个真正能够使用的 Audio 硬件抽象层，但是它需要 Android 的一种特殊的声音驱动程序支持。

在通用硬件抽象层中，类 AudioStreamOutGeneric、AudioStreamInGeneric 和 AudioHardwareGeneric 分别继承 Audio 硬件抽象层的 3 个接口。对应代码如下所示。

```
class AudioStreamOutGeneric : public AudioStreamOut {
    //...通用 Audio 输出类的接口
};
class AudioStreamInGeneric : public AudioStreamIn {
    //...通用 Audio 输入类的接口
};
class AudioHardwareGeneric : public AudioHardwareBase
{
    //...通用 Audio 控制类的接口
};
```

在文件 AudioHardwareGeneric.cpp 中使用的驱动程序是/dev/eac，这是一个非标准程序，定义设备路径的代码如下所示。

```
static char const * const kAudioDeviceName = "/dev/eac";
```

注意： eac 是 Linux 中的一个 misc 驱动程序，作为 Android 的通用音频驱动，写设备表示放音，读设备表示录音。

在 Linux 操作系统中，/dev/eac 驱动程序在文件系统中的节点主设备号为 10，是次设备号自动生成的。通过构造函数 AudioHardwareGeneric()可以打开这个驱动程序的设备节点。对应代码如下所示。

```
AudioHardwareGeneric::AudioHardwareGeneric()
    : mOutput(0), mInput(0), mFd(-1), mMicMute(false)
{
    mFd = ::open(kAudioDeviceName, O_RDWR);            //打开通用音频设备的节点
}
```

此音频设备是一个比较简单的驱动程序，在里面并没有很多设置接口，只是用写设备来表示录音，用读设备来表示放音。放音和录音支持的都是 16 位的 PCM，对应的实现代码如下所示。

```
ssize_t AudioStreamOutGeneric::write(const void* buffer, size_t bytes)
{
    Mutex::Autolock _l(mLock);
    return ssize_t(::write(mFd, buffer, bytes));            //写入硬件设备
}
ssize_t AudioStreamInGeneric::read(void* buffer, ssize_t bytes)
{
    AutoMutex lock(mLock);
    if (mFd < 0) {
        return NO_INIT;
    }
    return ::read(mFd, buffer, bytes);                      //读取硬件设备
}
```

尽管 AudioHardwareGeneric 是一个可以真正工作的 Audio 硬件抽象层，但是这种实现方式非常简单，不支持各种设置，参数也只能使用默认的。而且这种驱动程序需要在 Linux 核心加入 eac 驱动程序的支持。

4．具备 Dump 功能的 Audio 硬件抽象层

在文件 AudioDumpInterface.h 和 AudioDumpInterface.cpp 中，提供了具备 Dump 功能的 Audio 硬件抽象层，目的是将输出的 Audio 数据写入文件中。

其实 AudioDumpInterface 本身支持 Audio 输出功能，但是不支持输入功能。在文件 AudioDumpInterface.h 中定义类的代码如下所示。

```
class AudioStreamOutDump : public AudioStreamOut {
public:
                        AudioStreamOutDump( AudioStreamOut* FinalStream);
                        ~AudioStreamOutDump();
                        virtual ssize_t write(const void* buffer, size_t bytes);
    virtual uint32_t    sampleRate() const { return mFinalStream->sampleRate(); }
    virtual size_t      bufferSize() const { return mFinalStream->bufferSize(); }
    virtual int         channelCount() const { return
 mFinalStream->channelCount(); }
    virtual int         format() const { return mFinalStream->format(); }
    virtual uint32_t    latency() const { return   mFinalStream->latency(); }
    virtual status_t    setVolume(float volume)
      { return   mFinalStream->setVolume(volume); }
    virtual status_t    standby();
};
class AudioDumpInterface : public AudioHardwareBase
{
    virtual AudioStreamOut* openOutputStream(
```

```
                                int format=0,
                                int channelCount=0,
                                uint32_t sampleRate=0,
                                status_t *status=0);
}
```

在上述代码中，只实现了 AudioStreamOut 输出，而没有实现 AudioStreamIn 输入。由此可见，此 Audio 硬件抽象层只支持输出功能，不支持输入功能。其中输出文件的名称被定义为如下格式。

```
#define FLINGER_DUMP_NAME "/data/FlingerOut.pcm"
```

在文件 AudioDumpInterface.cpp 中，通过函数 AudioStreamOut()实现写操作，写入的对象就是这个文件，对应的实现代码如下所示。

```
ssize_t AudioStreamOutDump::write(const void* buffer, size_t bytes)
{
    ssize_t ret;
    ret = mFinalStream->write(buffer, bytes);
    if(!mOutFile && gFirst) {
        gFirst = false;
        mOutFile = fopen(FLINGER_DUMP_NAME, "r");
        if(mOutFile) {
            fclose(mOutFile);
            mOutFile = fopen(FLINGER_DUMP_NAME, "ab");
//打开输出文件
        }
    }
    if (mOutFile) {
        fwrite(buffer, bytes, 1, mOutFile);
//写文件输出内容
    }
    return ret;
}
```

如果文件是打开的，则可以使用追加方式写入。当使用这个 Audio 硬件抽象层时，播放的内容（PCM）将全部被写入文件。而且这个类支持各种格式的输出，具体什么格式将取决于调用者的设置。

使用 AudioDumpInterface 的目的并不是为了实际的应用，而是为了调试我们使用的类。当使用播放器调试音频时，有时无法确认是解码器的问题还是 Audio 输出单元的问题，这时就可以用这个类来替换实际的 Audio 硬件抽象层，将解码器输出的 Audio 的 PCM 数据写入文件中，由此可以判断解码器的输出是否正确。

19.3.4　真正实现 Audio 硬件抽象层

想要实现一个真正的 Audio 硬件抽象层，需要完成和 19.2 节中实现硬件抽象层类似的工作。例如可以基于 Linux 标准的音频驱动 OSS（Open Sound System）或 ALSA（Advanced Linux Sound Architecture）驱动程序来实现。

（1）基于 OSS 驱动程序实现

对于 OSS 驱动程序来说，实现方式和前面的 AudioHardwareGeneric 方式类似，数据流的读/写操作通过对/dev/dsp 设备的读/写来完成；区别在于 OSS 支持了更多的 ioctl 来进行设置，还涉及通过/dev/mixer 设备进行控制，并支持更多不同的参数。

（2）ALSA 驱动程序

对于 ALSA 驱动程序来说，实现方式一般不是直接调用驱动程序的设备节点，而是先实现用户空间的 alsa-lib，然后 Audio 硬件抽象层通过调用 alsa-lib 来实现。

在实现 Audio 硬件抽象层时，如果系统中有多个 Audio 设备，此时可由硬件抽象层自行处理 setRouting()函数设定。例如可以选择支持多个设备的同时输出，或者有优先级输出。对于这种情况，数据流一般来自函数 AudioStreamOut::write()，可由硬件抽象层确定输出方法。对于某种特殊的情况，也有可能采用硬件直接连接的方式，此时数据流可能并不来自上面的 write()，这样就没有数据通道，只有控制接口。Audio 硬件抽象层也是可以处理这种情况的。

19.4 实战演练——在 MSM 平台实现 Audio 驱动

经过本章前面内容的学习，读者已经基本了解了 Android 系统中 Audio 驱动程序的基本架构和移植知识。本节将讲解 MSM 平台中 Audio 系统的实现方法。

19.4.1 实现 Audio 驱动程序

在 MSM 平台中，Audio 驱动程序被保存在 arch/arm/mach-msm/qdspX 目录中。如果是版本为 5 的 DSP 处理器，其驱动目录为 qdsp5；如果是版本为 6 的 DSP 处理器，其驱动目录为 qdsp6。Audio 驱动和 MSM 处理器的 DSP 系统是密切相关的。

通常 Audio 驱动程序的头文件是/include/linux/msm_mdp.h，而 qdsp6 的特定头文件是 arch/arm/mach-msm/include/mach/msm_qdsp6_audio.h。在 Audio 系统中涉及的头文件如下。

☑ audio_ctl.c：音频控制文件，生成设备的节点是 dev/msm_audio_ctl。
☑ routing.c：控制音频路径，生成设备的节点是 dev/msm_audio_route。
☑ pcm_in.c：PCM 输入通道，生成设备的节点是 dev/msm_pcm_out。
☑ mp3.c：MP3 码流直接输出通道，生成设备的节点是 dev/msm_mp3。

在 MSM 平台中，Audio 驱动程序并不是主流的标准驱动程序，在用户空间中包括了两个控制节点和 3 个数据节点。其中两个控制节点用于控制 Audio 的基本内容和路径，而 3 个数据节点包括 PCM 输出、PCM 输入和 MP3 码流输出。

在文件 include/linux/msm_audio.h 中定义了 Audio 系统的 ioctl 控制命令，具体代码如下所示。

```
#define AUDIO_IOCTL_MAGIC 'a'

 #define AUDIO_START              _IOW(AUDIO_IOCTL_MAGIC, 0, unsigned)
 #define AUDIO_STOP               _IOW(AUDIO_IOCTL_MAGIC, 1, unsigned)
 #define AUDIO_FLUSH              _IOW(AUDIO_IOCTL_MAGIC, 2, unsigned)
 #define AUDIO_GET_CONFIG         _IOR(AUDIO_IOCTL_MAGIC, 3, unsigned)
 #define AUDIO_SET_CONFIG         _IOW(AUDIO_IOCTL_MAGIC, 4, unsigned)
 #define AUDIO_GET_STATS          _IOR(AUDIO_IOCTL_MAGIC, 5, unsigned)
 #define AUDIO_ENABLE_AUDPP _IOW(AUDIO_IOCTL_MAGIC, 6, unsigned)
 #define AUDIO_SET_ADRC           _IOW(AUDIO_IOCTL_MAGIC, 7, unsigned)
 #define AUDIO_SET_EQ             _IOW(AUDIO_IOCTL_MAGIC, 8, unsigned)
 #define AUDIO_SET_RX_IIR         _IOW(AUDIO_IOCTL_MAGIC, 9, unsigned)
 #define AUDIO_SET_VOLUME         _IOW(AUDIO_IOCTL_MAGIC, 10, unsigned)
 #define AUDIO_PAUSE              _IOW(AUDIO_IOCTL_MAGIC, 11, unsigned)
 #define AUDIO_PLAY_DTMF          _IOW(AUDIO_IOCTL_MAGIC, 12, unsigned)
 #define AUDIO_GET_EVENT          _IOR(AUDIO_IOCTL_MAGIC, 13, unsigned)
 #define AUDIO_ABORT_GET_EVENT _IOW(AUDIO_IOCTL_MAGIC, 14, unsigned)
```

```
......
#define AUDIO_MAX_COMMON_IOCTL_NUM         100
```

19.4.2　实现硬件抽象层

在 MSM 平台中，硬件抽象层已经包含在 Android 的开放源码中，这些都是通用的代码。这些代码被保存在目录 hardware/msm7k 下，使用 MSM7K 处理器实现 libaudio，通过 QSD8K 处理器实现 libaudio-qsd8k。其中在 libaudio-qsd8k 目录下主要包含如下。

☑ AudioHardware.cpp：实现了 Audio 硬件抽象层。
☑ AudioHardware.h：定义了 Audio 硬件抽象层的类。
☑ AudioPolicyManager.cpp：实现了 Audio 策略管理。
☑ AudioPolicyManager.h：定义了 Audio 策略管理类。
☑ msm_audio.h：实现了和内核相同的 ioctl 命令和数据结构的定义。

在文件 AudioHardware.h 中定义了 3 个类，分别是 AudioHardware、class AudioStreamOutMSM72xx 和 AudioStreamInMSM72xx，这 3 个类都继承自其他类，分别实现 Audio 系统的总控、输出和输入环节的功能，具体代码如下所示。

```
class AudioHardware : public   AudioHardwareBase
{
    class AudioStreamOutMSM72xx;
    class AudioStreamInMSM72xx;
public:
                        AudioHardware();
    virtual             ~AudioHardware();
    virtual status_t    initCheck();
class AudioStreamOutMSM72xx : public AudioStreamOut {
    public:
                            AudioStreamOutMSM72xx();
        virtual             ~AudioStreamOutMSM72xx();
                status_t    set(AudioHardware* mHardware,
                                uint32_t devices,
                                int *pFormat,
                                uint32_t *pChannels,
                                uint32_t *pRate);
..........................................
  private:
                AudioHardware* mHardware;
                int         mFd;
                int         mStartCount;
                int         mRetryCount;
                bool        mStandby;
                uint32_t    mDevices;
    };
    class AudioStreamInMSM72xx : public AudioStreamIn {
    public:
        enum input_state {
        AUDIO_INPUT_CLOSED,
```

```
            AUDIO_INPUT_OPENED,
            AUDIO_INPUT_STARTED
        };
......................................
```

类 AudioStreamOutMSM72xx 的核心功能是通过函数 write()实现的，此函数的实现代码如下所示。

```
ssize_t AudioHardware::AudioStreamOutMSM72xx::write(const void* buffer, size_t bytes)
{
    status_t status = NO_INIT;
    size_t count = bytes;
    const uint8_t* p = static_cast<const uint8_t*>(buffer);
    if (mStandby) {
        LOGV("open pcm_out driver");
        status = ::open("/dev/msm_pcm_out", O_RDWR);//打开驱动程序
        if (status < 0) {
            if (errCount++ < 10) {
                LOGE("Cannot open /dev/msm_pcm_out errno: %d", errno);
            }
            goto Error;
        }
        mFd = status;
        LOGV("get config");
        struct msm_audio_config config;
        status = ioctl(mFd, AUDIO_GET_CONFIG, &config);//获取配置
        if (status < 0) {
            LOGE("Cannot read pcm_out config");
            goto Error;
        }
        LOGV("set pcm_out config");
        config.channel_count = AudioSystem::popCount(channels());
        config.sample_rate = sampleRate();
        config.buffer_size = bufferSize();
        config.buffer_count = AUDIO_HW_NUM_OUT_BUF;
        config.codec_type = CODEC_TYPE_PCM;
        status = ioctl(mFd, AUDIO_SET_CONFIG, &config);//开始进行配置
        if (status < 0) {
            LOGE("Cannot set config");
            goto Error;
        }
        LOGV("buffer_size: %u", config.buffer_size);
        LOGV("buffer_count: %u", config.buffer_count);
        LOGV("channel_count: %u", config.channel_count);
        LOGV("sample_rate: %u", config.sample_rate);
        uint32_t acdb_id = mHardware->getACDB(MOD_PLAY, mHardware->get_snd_dev());
        status = ioctl(mFd, AUDIO_START, &acdb_id);//开始 Audio
        if (status < 0) {
            LOGE("Cannot start pcm playback");
            goto Error;
        }
        status = ioctl(mFd, AUDIO_SET_VOLUME, &stream_volume);//设置音量
        if (status < 0) {
            LOGE("Cannot start pcm playback");
```

```
                goto Error;
            }
            LOGV("acquire wakelock");
            acquire_wake_lock(PARTIAL_WAKE_LOCK, kOutputWakelockStr);
            mStandby = false;
        }
        while (count) {
            ssize_t written = ::write(mFd, p, count);//写操作 PCM 输出
            if (written >= 0) {//计算剩余数据
                count -= written;
                p += written;
            } else {
                if (errno != EAGAIN) return written;
                mRetryCount++;
                LOGW("EAGAIN - retry");
            }
        }
        return bytes;
Error:
        if (mFd >= 0) {
            ::close(mFd);
            mFd = -1;
        }
        usleep(bytes * 1000000 / frameSize() / sampleRate());
        return status;
}
```

上述函数的处理过程是，先打开 PCM 设备来输出配置，然后设置配置，并通过 ioctl 命令开始 Audio 处理流，并读、写操作文件。

类 AudioStreamInMSM72xx 的核心功能是通过函数 read()实现的，此函数的实现代码如下所示。

```
ssize_t AudioHardware::AudioStreamInMSM72xx::read( void* buffer, ssize_t bytes)
{
        LOGV("AudioStreamInMSM72xx::read(%p, %ld)", buffer, bytes);
        if (!mHardware) return -1;
        size_t count = bytes;
        uint8_t* p = static_cast<uint8_t*>(buffer);
        if (mState < AUDIO_INPUT_OPENED) {
            Mutex::Autolock lock(mHardware->mLock);
            if (set(mHardware, mDevices, &mFormat, &mChannels, &mSampleRate, mAcoustics) != NO_ERROR) {
                return -1;
            }
        }
        if (mState < AUDIO_INPUT_STARTED) {
            mHardware->set_mRecordState(1);
            if (support_a1026 == 1) {
                mHardware->doAudience_A1026_Control(mHardware->get_mMode(), 1, mHardware->get_snd_dev());
            }
            uint32_t acdb_id = mHardware->getACDB(MOD_REC, mHardware->get_snd_dev());
            if (ioctl(mFd, AUDIO_START, &acdb_id)) {                //开始 Audio 数据流
                LOGE("Error starting record");
                return -1;
```

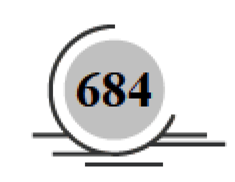
```
            }
            LOGI("AUDIO_START: start kernel pcm_in driver.");
            mState = AUDIO_INPUT_STARTED;
        }
        while (count) {
            ssize_t bytesRead = ::read(mFd, buffer, count);          //读操作
            if (bytesRead >= 0) {                                    //计算出还需要读取的数据量
                count -= bytesRead;
                p += bytesRead;
            } else {
                if (errno != EAGAIN) return bytesRead;
                mRetryCount++;
                LOGW("EAGAIN - retrying");
            }
        }
        return bytes;
}
```

在上述代码中，通过 mFd 打开 Audio 输入设备的描述，输入设备是 dev/msm/_pcm_in。其实在函数 set()中已经完成了对设备 dev/msm/_pcm_in 的基本配置工作，函数 set()的具体实现代码如下所示。

```
status_t AudioHardware::AudioStreamOutMSM72xx::set(
            AudioHardware* hw, uint32_t devices, int *pFormat, uint32_t *pChannels, uint32_t *pRate)
{
        int lFormat = pFormat ? *pFormat : 0;
        uint32_t lChannels = pChannels ? *pChannels : 0;
        uint32_t lRate = pRate ? *pRate : 0;
        mHardware = hw;
        if (lFormat == 0) lFormat = format();
        if (lChannels == 0) lChannels = channels();
        if (lRate == 0) lRate = sampleRate();
        if ((lFormat != format()) ||
            (lChannels != channels()) ||
            (lRate != sampleRate())) {
            if (pFormat) *pFormat = format();
            if (pChannels) *pChannels = channels();
            if (pRate) *pRate = sampleRate();
            return BAD_VALUE;
        }
        if (pFormat) *pFormat = lFormat;
        if (pChannels) *pChannels = lChannels;
        if (pRate) *pRate = lRate;
        mDevices = devices;
      return NO_ERROR;
}
```

19.5 实战演练——在 OSS 平台实现 Audio 驱动

OSS（Open Sound System）是 UNIX 平台上一个统一的音频接口。本节将详细讲解在 OSS 平台上实现 Audio 系统的基本知识。



19.5.1　OSS 驱动基础

OSS 驱动是字符型设备，因为在 UNIX 系统中所有的设备都被统一成文件，通过对文件的访问方式（首先 open，然后 read/write，同时可以使用 ioctl 读取/设置参数，最后 close）来访问设备。所以在 OSS 中主要包含以下几种设备文件。

- ☑ /dev/mixer：访问声卡中内置的 mixer，调整音量大小，选择音源。
- ☑ /dev/sndstat：测试声卡，执行 cat/dev/sndstat 会显示声卡驱动的信息。
- ☑ /dev/dsp、/dev/dspW 和/dev/audio：读这个设备就相当于录音，写这个设备就相当于放音。/dev/dsp 与/dev/audio 之间的区别在于采样的编码不同，/dev/audio 使用 μ 律编码，/dev/dsp 使用 8-bit（无符号）线形编码，/dev/dspW 使用 16-bit（有符号）线形编码。/dev/audio 主要是为了与 SunOS 兼容，所以尽量不要使用。
- ☑ /dev/sequencer：访问声卡内置的，或者连接在 MIDI 接口的 synthesizer。

在 Linux 系统中，有如下 3 个和 OSS 相关的文件。

- ☑ include/linux/sound.h
- ☑ sound/sound_core.c
- ☑ include/linux/soundcard.h

Linux 的音频输入、输出是通过/dev/dsp 设备实现的，但对于这些声音信号的处理则是通过/dev/mixer 设备来完成的。查看文件 linux/soundcard.h 可以获取对 Mixer 文件操作所需要的变量，在此文件中列出了如下常用的变量。

- ☑ SOUND_MIXER_WRITE_VOLUME = 0xc0044d00
- ☑ SOUND_MIXER_WRITE_BASS = 0xc0044d01
- ☑ SOUND_MIXER_WRITE_PCM = 0xc0044d04
- ☑ SOUND_MIXER_WRITE_LINE = 0xc0044d06
- ☑ SOUND_MIXER_WRITE_MIC = 0xc0044d07
- ☑ SOUND_MIXER_WRITE_RECSRC = 0xc0044dff
- ☑ SOUND_MIXER_LINE = 7
- ☑ SOUND_MASK_LINE = 64

上述变量名都可以在文件 soundcard.h 中查到，通过名称即可看出其用途，后面的变量赋值在该头文件中并不是这样定义的，而是通过调用一些函数返回来的，应该是声卡上对应的地址。在应用程序中可通过 ioctl(fd,cmd,arg)对这些变量进行赋值。其中 fd 即为一个打开/dev/mixer 的文件指针，cmd 为上面所列的这些变量，arg 即是对这些变量进行操作所需赋给的结构体或变量。

19.5.2　函数 mixer()

在 OSS 平台中，函数 mixer()是核心，功能是缩进一组控制音频线到目标设备的函数，并且可以控制音量和其他效果。在这组 API 中，尽管只有 10 个函数和两个消息，但使用起来还是比较难。本节将通过应用这些函数编写成两个应用程序来展示它们的使用方法，而且尽可能采用实际应用中的用户界面，只有这样才更有可能被读者直接使用。

1．程序 1

光盘:\daima\19\MIXER_MUTE

此程序运行后能够等效于 Windows Volume Control 的 Mute all 核选框，核心功能是通过文件 MUTEDLG.CPP 实现的，主要代码如下所示。

```
LONG CMuteDlg::OnMixerCtrlChange(UINT wParam, LONG lParam)
{
	if ((HMIXER)wParam == m_hMixer && (DWORD)lParam == m_dwMuteControlID)
	{
		// The state of the master mute control has changed. Refresh it.
		LONG lVal;
		if (this->amdGetMasterMuteValue(lVal))
			m_bMute = lVal;

		this->UpdateData(FALSE);
	}

	return 0L;
}

BOOL CMuteDlg::amdInitialize()
{
	// get the number of mixer devices present in the system
	m_nNumMixers = ::mixerGetNumDevs();

	m_hMixer = NULL;
	::ZeroMemory(&m_mxcaps, sizeof(MIXERCAPS));

	// open the first mixer
	// A "mapper" for audio mixer devices does not currently exist
	if (m_nNumMixers != 0)
	{
		if (::mixerOpen(&m_hMixer,
						0,
						(DWORD)this->GetSafeHwnd()
						NULL,
						MIXER_OBJECTF_MIXER | CALLBACK_WINDOW)
			!= MMSYSERR_NOERROR)
			return FALSE;

		if (::mixerGetDevCaps((UINT)m_hMixer, &m_mxcaps, sizeof(MIXERCAPS))
			!= MMSYSERR_NOERROR)
			return FALSE;
	}

	return TRUE;
}

BOOL CMuteDlg::amdUninitialize()
{
	BOOL bSucc = TRUE;

	if (m_hMixer != NULL)
```

```
    {
        bSucc = ::mixerClose(m_hMixer) == MMSYSERR_NOERROR;
        m_hMixer = NULL;
    }

    return bSucc;
}

BOOL CMuteDlg::amdGetMasterMuteControl()
{
    m_strDstLineName.Empty();
    m_strMuteControlName.Empty();

    if (m_hMixer == NULL)
        return FALSE;

    // get dwLineID
    MIXERLINE mxl;
    mxl.cbStruct = sizeof(MIXERLINE);
    mxl.dwComponentType = MIXERLINE_COMPONENTTYPE_DST_SPEAKERS;
    if (::mixerGetLineInfo((HMIXEROBJ)m_hMixer,
                           &mxl,
                           MIXER_OBJECTF_HMIXER |
                           MIXER_GETLINEINFOF_COMPONENTTYPE)
        != MMSYSERR_NOERROR)
        return FALSE;

    // get dwControlID
    MIXERCONTROL mxc;
    MIXERLINECONTROLS mxlc;
    mxlc.cbStruct = sizeof(MIXERLINECONTROLS);
    mxlc.dwLineID = mxl.dwLineID;
    mxlc.dwControlType = MIXERCONTROL_CONTROLTYPE_MUTE;
    mxlc.cControls = 1;
    mxlc.cbmxctrl = sizeof(MIXERCONTROL);
    mxlc.pamxctrl = &mxc;
    if (::mixerGetLineControls((HMIXEROBJ)m_hMixer,
                               &mxlc,
                               MIXER_OBJECTF_HMIXER |
                               MIXER_GETLINECONTROLSF_ONEBYTYPE)
        != MMSYSERR_NOERROR)
        return FALSE;

    // record dwControlID
    m_strDstLineName = mxl.szName;
    m_strMuteControlName = mxc.szName;
    m_dwMuteControlID = mxc.dwControlID;

    return TRUE;
}
```

```
BOOL CMuteDlg::amdGetMasterMuteValue(LONG &lVal) const
{
	if (m_hMixer == NULL ||
		m_strDstLineName.IsEmpty() || m_strMuteControlName.IsEmpty())
		return FALSE;

	MIXERCONTROLDETAILS_BOOLEAN mxcdMute;
	MIXERCONTROLDETAILS mxcd;
	mxcd.cbStruct = sizeof(MIXERCONTROLDETAILS);
	mxcd.dwControlID = m_dwMuteControlID;
	mxcd.cChannels = 1;
	mxcd.cMultipleItems = 0;
	mxcd.cbDetails = sizeof(MIXERCONTROLDETAILS_BOOLEAN);
	mxcd.paDetails = &mxcdMute;
	if (::mixerGetControlDetails((HMIXEROBJ)m_hMixer,
							   &mxcd,
							   MIXER_OBJECTF_HMIXER |
							   MIXER_GETCONTROLDETAILSF_VALUE)
		!= MMSYSERR_NOERROR)
		return FALSE;

	lVal = mxcdMute.fValue;

	return TRUE;
}

BOOL CMuteDlg::amdSetMasterMuteValue(LONG lVal) const
{
	if (m_hMixer == NULL ||
		m_strDstLineName.IsEmpty() || m_strMuteControlName.IsEmpty())
		return FALSE;

	MIXERCONTROLDETAILS_BOOLEAN mxcdMute = { lVal };
	MIXERCONTROLDETAILS mxcd;
	mxcd.cbStruct = sizeof(MIXERCONTROLDETAILS);
	mxcd.dwControlID = m_dwMuteControlID;
	mxcd.cChannels = 1;
	mxcd.cMultipleItems = 0;
	mxcd.cbDetails = sizeof(MIXERCONTROLDETAILS_BOOLEAN);
	mxcd.paDetails = &mxcdMute;
	if (::mixerSetControlDetails((HMIXEROBJ)m_hMixer,
							   &mxcd,
							   MIXER_OBJECTF_HMIXER |
							   MIXER_SETCONTROLDETAILSF_VALUE)
		!= MMSYSERR_NOERROR)
		return FALSE;

	return TRUE;
}
```

执行效果如图 19-2 所示。



图 19-2　执行效果

2．程序 2

光盘:\daima\19\MIXER_VOLUME

此程序运行后能够等效于 Windows Volume Control 的进度条，核心功能是通过文件 VOLUMEDLG.CPP 实现的，主要实现代码如下所示。

```
LONG CVolumeDlg::OnMixerCtrlChange(UINT wParam, LONG lParam)
{
	if ((HMIXER)wParam == m_hMixer && (DWORD)lParam == m_dwVolumeControlID)
	{
		// The state of the master volume control has changed. Refresh it
		DWORD dwVal;
		if (this->amdGetMasterVolumeValue(dwVal))
			m_ctrlSlider.SetPos(dwVal);
	}

	return 0L;
}

BOOL CVolumeDlg::amdInitialize()
{
	// get the number of mixer devices present in the system
	m_nNumMixers = ::mixerGetNumDevs();

	m_hMixer = NULL;
	::ZeroMemory(&m_mxcaps, sizeof(MIXERCAPS));

	// open the first mixer
	// A "mapper" for audio mixer devices does not currently exist
	if (m_nNumMixers != 0)
	{
		if (::mixerOpen(&m_hMixer,
						0,
						(DWORD)this->GetSafeHwnd(),
						NULL,
						MIXER_OBJECTF_MIXER | CALLBACK_WINDOW)
			!= MMSYSERR_NOERROR)
			return FALSE;

		if (::mixerGetDevCaps((UINT)m_hMixer, &m_mxcaps, sizeof(MIXERCAPS))
			!= MMSYSERR_NOERROR)
			return FALSE;
```

```
	}

	return TRUE;
}

BOOL CVolumeDlg::amdUninitialize()
{
	BOOL bSucc = TRUE;

	if (m_hMixer != NULL)
	{
		bSucc = ::mixerClose(m_hMixer) == MMSYSERR_NOERROR;
		m_hMixer = NULL;
	}

	return bSucc;
}

BOOL CVolumeDlg::amdGetMasterVolumeControl()
{
	m_strDstLineName.Empty();
	m_strVolumeControlName.Empty();

	if (m_hMixer == NULL)
		return FALSE;

	// get dwLineID
	MIXERLINE mxl;
	mxl.cbStruct = sizeof(MIXERLINE);
	mxl.dwComponentType = MIXERLINE_COMPONENTTYPE_DST_SPEAKERS;
	if (::mixerGetLineInfo((HMIXEROBJ)m_hMixer,
	                       &mxl,
	                       MIXER_OBJECTF_HMIXER |
	                       MIXER_GETLINEINFOF_COMPONENTTYPE)
		!= MMSYSERR_NOERROR)
		return FALSE;

	// get dwControlID
	MIXERCONTROL mxc;
	MIXERLINECONTROLS mxlc;
	mxlc.cbStruct = sizeof(MIXERLINECONTROLS);
	mxlc.dwLineID = mxl.dwLineID;
	mxlc.dwControlType = MIXERCONTROL_CONTROLTYPE_VOLUME;
	mxlc.cControls = 1;
	mxlc.cbmxctrl = sizeof(MIXERCONTROL);
	mxlc.pamxctrl = &mxc;
	if (::mixerGetLineControls((HMIXEROBJ)m_hMixer,
	                           &mxlc,
	                           MIXER_OBJECTF_HMIXER |
	                           MIXER_GETLINECONTROLSF_ONEBYTYPE)
		!= MMSYSERR_NOERROR)
```

```
        return FALSE;

    // record dwControlID
    m_strDstLineName = mxl.szName;
    m_strVolumeControlName = mxc.szName;
    m_dwMinimum = mxc.Bounds.dwMinimum;
    m_dwMaximum = mxc.Bounds.dwMaximum;
    m_dwVolumeControlID = mxc.dwControlID;

    return TRUE;
}

BOOL CVolumeDlg::amdGetMasterVolumeValue(DWORD &dwVal) const
{
    if (m_hMixer == NULL ||
        m_strDstLineName.IsEmpty() || m_strVolumeControlName.IsEmpty())
        return FALSE;

    MIXERCONTROLDETAILS_UNSIGNED mxcdVolume;
    MIXERCONTROLDETAILS mxcd;
    mxcd.cbStruct = sizeof(MIXERCONTROLDETAILS);
    mxcd.dwControlID = m_dwVolumeControlID;
    mxcd.cChannels = 1;
    mxcd.cMultipleItems = 0;
    mxcd.cbDetails = sizeof(MIXERCONTROLDETAILS_UNSIGNED);
    mxcd.paDetails = &mxcdVolume;
    if (::mixerGetControlDetails((HMIXEROBJ)m_hMixer,
                                &mxcd,
                                MIXER_OBJECTF_HMIXER |
                                MIXER_GETCONTROLDETAILSF_VALUE)
        != MMSYSERR_NOERROR)
        return FALSE;

    dwVal = mxcdVolume.dwValue;

    return TRUE;
}

BOOL CVolumeDlg::amdSetMasterVolumeValue(DWORD dwVal) const
{
    if (m_hMixer == NULL ||
        m_strDstLineName.IsEmpty() || m_strVolumeControlName.IsEmpty())
        return FALSE;

    MIXERCONTROLDETAILS_UNSIGNED mxcdVolume = { dwVal };
    MIXERCONTROLDETAILS mxcd;
    mxcd.cbStruct = sizeof(MIXERCONTROLDETAILS);
    mxcd.dwControlID = m_dwVolumeControlID;
    mxcd.cChannels = 1;
    mxcd.cMultipleItems = 0;
    mxcd.cbDetails = sizeof(MIXERCONTROLDETAILS_UNSIGNED);
```

```
        mxcd.paDetails = &mxcdVolume;
        if (::mixerSetControlDetails((HMIXEROBJ)m_hMixer,
                                                &mxcd,
                                                MIXER_OBJECTF_HMIXER |
                                                MIXER_SETCONTROLDETAILSF_VALUE)
            != MMSYSERR_NOERROR)
            return FALSE;

        return TRUE;
}
```

执行效果如图 19-3 所示。



图 19-3　执行效果

19.6　实战演练——在 ALSA 平台实现 Audio 系统

ALSA 是 Advanced Linux Sound Architecture 的缩写，是高级 Linux 声音架构的简称。ALSA 在 Linux 操作系统上提供了音频和 MIDI（Musical Instrument Digital Interface，音乐设备数字化接口）的支持，从 2.6 系列内核开始，ALSA 即成为默认的声音子系统，用来替换 2.4 系列内核中的 OSS（Open Sound System，开放声音系统），本节将详细讲解在 ALSA 平台上实现 Audio 系统的基本知识。

19.6.1　注册音频设备和音频驱动

1．注册音频设备

在设备的 drvdata 中包含了 3 部分，分别是关于 machine、Platform 和 codec 的。大体上，machine 主要是关于 CPU，也可以说是关于 SSP 本身设置的；而 Platform 是关于平台级别的，即和这个平台本身实现相关的；而 codec 就是和我们所用的音频 codec 相关的。整个 ALSA 音频驱动的架构特点是从 ALSA 层进入→内核 ALSA 层接口→core 层，在上述流程中分别调用这 3 个方面的函数来处理，先是 CPU 级别，然后是 Platform，最后是 codec 级别。

2．注册音频驱动

前面讲了设备的注册，里面的设备名字就是 soc-audio，而这里的 driver 的注册时名字也是 soc-audio，对于 Platform 的设备匹配原则是根据名字进行的，所以将会匹配成功，成功后就会执行 Audio 驱动提供的函数 soc_probe()。

函数 soc_probe()本身架构很简单，先调用了 CPU 级别的 Probe，然后是 codec 级别的，最后是 Platform 的，这里 3 个的顺序不一样，但是因为 CPU 级别和 Platform 级别都为空，最后都调用了 codec 级别的 Probe



函数，也就是 micco_soc_probe()。函数 micco_soc_probe()基本上能够完成所有应该完成的音频驱动的初始化。

19.6.2　在 Android 中使用 ALSA 声卡

在 Android 系统中，可以使用 ALSA 声卡来实现音频效果功能。

1．使用过程

在 Android 系统中，使用 ALSA 声卡的具体流程如下。

（1）使用下面的 CD 命令来到 Android 源码的根目录，在此以 Android 4.3 为例。

```
cd /home/figo/android/Android-4.3
```

（2）从 Android 主页下载 ALSA 声卡有关源码，下载命令如下所示。

```
git clone git://android.git.kernel.org/platform/external/alsa-lib.git
git clone git://android.git.kernel.org/platform/external/alsa-utils.git
git clone git://android.git.kernel.org/platform/hardware/alsa_sound.git
```

（3）下载完成后修改板子的 BoardConfig.mk 文件，确保板子可以使用 ALSA 声卡，修改代码如下所示。

```
#HAVE_HTC_AUDIO_DRIVER := true
#BOARD_USES_GENERIC_AUDIO := true
BOARD_USES_ALSA_AUDIO := true
BUILD_WITH_ALSA_UTILS := true
```

在 Android-2.0 版本中，文件 BoardConfig.mk 位于目录 Android-2.0/build/target/board/generic 中。

（4）重新编译一遍 Android，编译完成后在根文件/system/etc/asound.conf 中添加配置声卡的工作参数脚本，具体代码如下所示。

```
##
## Mixer devices
##
ctl.AndroidPlayback {
type hw
card 0 # Can replace with drivers name from /proc/asound/cards
}
ctl.AndroidRecord {
type hw
card 0
}

##
## Playback devices
##
pcm.AndroidPlayback {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Speaker {
type hw
card 0
device 0
}
```

```
pcm.AndroidPlayback_Speaker_normal {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Speaker_ringtone {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Speaker_incall {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Earpiece {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Earpiece_normal {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Earpiece_ringtone {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Earpiece_incall {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Bluetooth {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Bluetooth_normal {
type hw
```

```
card 0
device 0
}

pcm.AndroidPlayback_Bluetooth_ringtone {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Bluetooth_incall {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Headset {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Headset_normal {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Headset_ringtone {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Headset_incall {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Bluetooth-A2DP {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Bluetooth-A2DP_normal {
type hw
card 0
device 0
}
```

```
pcm.AndroidPlayback_Bluetooth-A2DP_ringtone {
type hw
card 0
device 0
}

pcm.AndroidPlayback_Bluetooth-A2DP_incall {
type hw
card 0
device 0
}

pcm.AndroidRecord {
type hw
card 0
device 0
}

pcm.AndroidRecord_Microphone {
type hw
card 0
device 0
}
```

2．几个需要注意的问题

（1）ALSA 音频路径

在 sound/soc/codecs 目录下保存了很多音频的 Codec 驱动，例如笔者使用的是 wm9713，AP 是 S3C6410，在此驱动文件中定义了很多 widget 和 control，ALSA 在 PlayBack（回放）或 Record（录制）时，文件 sound/soc/soc-dapm.c 中的函数 dapm_power_widgets()会根据“配置情况”来打开相应的 Widget，并搭建一个完整的音频路径。只要搭建该音频路径成功，即可正常工作。

文件 sound/soc/codecs/wm9719.c 中的 audio_map[]就是一个 wm9713 的路由表，根据 wm9713 手册中的 Audio Paths Overview 可以选择自己需要的音频路径，可以在 audio_map[]中测试一下，看 audio_map 中是否支持这种路径。

（2）配置 ALSA

在 ALSA 中最主要的是配置 ALSA 音频调试，ALSA 使用 amixer 命令打开 audio_map[]中的开关（control/switch）和其他 control（控制）并设置这些 control，在使用“aplay（播放）/arecord（录音）”时即可搭建正确的路径，实现播放和录音功能。

例如在调试时，在不使用 amixer 控制时（这是默认状态），arecord 可以正确录音，使用文件 sound/soc/soc-dapm.c 中的函数 dump_dapm()来 Dump 出的路径是正确的；当 aplay 时，dump_dapm 出来的路径是错误的，原因是默认设置中没有打开 playback 的开关（switch）。当遇到上述问题时，可以运行如下命令即可正确地 playback。

```
alsa_amixer cset numid=4,iface=MIXER,name='Headphone Playback Switch' 1
alsa_amixer cset numid=93,iface=MIXER,name='Left Headphone Out Mux' 2
alsa_amixer cset numid=34,iface=MIXER,name='Out3 Playback Switch'1
alsa_amixer cset numid=95,iface=MIXER,name='Left Speaker Out Mux' 4
alsa_amixer cset numid=94,iface=MIXER,name='Right Speaker Out Mux' 2
```

```
alsa_amixer cset numid=91,iface=MIXER,name='Out 3 Mux' 2
alsa_amixer cset numid=81,iface=MIXER,name='Left HP Mixer PCM Playback Swit' 1
alsa_amixer cset numid=75,iface=MIXER,name='Right HP Mixer PCM Playback Swi' 1
alsa_amixer cset numid=3,iface=MIXER,name='Headphone Playback Volume' 26
alsa_amixer cset numid=36,iface=MIXER,name='Out3 Playback Volume' 48
```

由此可见，在打开 playback 路径时需要开关，dapm_power_widgets 会自动把这些开关连接的 widget 连接起来构成一个完整的播放路径。

（3）在 Android 中配置 ALSA

在 Android 中使用 alsa-lib 时也需要配置音频路径，具体来说有如下两个配置方法。

☑ 在文件 AudioHardwareALSA.cpp 中，使用函数 system()调用 amixer 来完成配置，具体代码如下所示。

```
system("alsa_amixer cset numid=4,iface=MIXER,name='Headphone Playback Switch' 1");
```

☑ 编写文件 asound.conf，在 AudioHardwareALSA.cpp 中的 ALSAMixer::ALSAMixer 对象初始化时，会通过 alsa-lib 的 conf.c 文件中的函数来读取文件/etc/asound.conf 以获取配置信息，并对 codec 进行配置。

3．ASLA 中的重要命令

（1）alsa_amixer 命令

命令 alsa_amixer 用于配置音频 codec 的 mixer 开关、mux 对路选择、volume 值等，例如下面的代码。

```
alsa_amixer --help
alsa_amixer contents
alsa_amixer contents
numid=30,iface=MIXER,name='Headphone Playback ZC Switch'
; type=BOOLEAN,access=rw------,values=2
: values=off,off
numid=4,iface=MIXER,name='Headphone Playback Switch'
; type=BOOLEAN,access=rw------,values=2
: values=off,off
numid=3,iface=MIXER,name='Headphone Playback Volume'
; type=INTEGER,access=rw------,values=2,min=0,max=31,step=0
: values=31,31
numid=6,iface=MIXER,name='PCM Playback Volume'
; type=INTEGER,access=rw------,values=2,min=0,max=31,step=0
: values=23,23
numid=5,iface=MIXER,name='Line In Volume'
; type=INTEGER,access=rw------,values=2,min=0,max=31,step=0
: values=23,23
numid=7,iface=MIXER,name='Mic 1 Volume'
; type=INTEGER,access=rw------,values=1,min=0,max=31,step=0
: values=23
numid=8,iface=MIXER,name='Mic 2 Volume'
; type=INTEGER,access=rw------,values=1,min=0,max=31,step=0
: values=23
numid=85,iface=MIXER,name='Mic A Source'
; type=ENUMERATED,access=rw------,values=1,items=3
; Item #0 'Mic 1'
; Item #1 'Mic 2 A'
; Item #2 'Mic 2 B'
: values=0
```

```
alsa_amixer controls
numid=30,iface=MIXER,name='Headphone Playback ZC Switch'
numid=4,iface=MIXER,name='Headphone Playback Switch'
numid=3,iface=MIXER,name='Headphone Playback Volume'
numid=6,iface=MIXER,name='PCM Playback Volume'
numid=5,iface=MIXER,name='Line In Volume'
numid=7,iface=MIXER,name='Mic 1 Volume'
numid=8,iface=MIXER,name='Mic 2 Volume'
numid=85,iface=MIXER,name='Mic A Source'
numid=84,iface=MIXER,name='Mic B Source'
numid=9,iface=MIXER,name='Mic Boost (+20dB) Switch'
numid=10,iface=MIXER,name='Mic Headphone Mixer Volume'
numid=47,iface=MIXER,name='Aux Playback Headphone Volume'
numid=48,iface=MIXER,name='Aux Playback Master Volume'
numid=49,iface=MIXER,name='Aux Playback Mono Volume'
numid=67,iface=MIXER,name='Mono Mixer Aux Playback Switch'
numid=69,iface=MIXER,name='Mono Mixer Bypass Playback Swit'
numid=70,iface=MIXER,name='Mono Mixer Mic 1 Sidetone Switc'
numid=71,iface=MIXER,name='Mono Mixer Mic 2 Sidetone Switc'
numid=65,iface=MIXER,name='Mono Mixer PC Beep Playback Swi'
numid=68,iface=MIXER,name='Mono Mixer PCM Playback Switch'
numid=66,iface=MIXER,name='Mono Mixer Voice Playback Switc'
```

（2）alsa_alsactl store 命令

此命令用于生成文件/etc/asound.state，在显示当前 codec 的状态时可以根据该文件检查 codec 的状态是否正确，例如下面的代码。

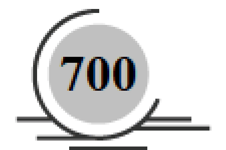
```
# cat /etc/asound.state
state.SMDK6400 {
control.1 {
comment.access 'read write'
comment.type INTEGER
comment.count 2
comment.range '0 - 31'
iface MIXER
name 'Speaker Playback Volume'
value.0 31
value.1 31
}
control.2 {
comment.access 'read write'
comment.type BOOLEAN
comment.count 2
iface MIXER
name 'Speaker Playback Switch'
value.0 false
value.1 false
}
control.3 {
comment.access 'read write'
comment.type INTEGER
```

```
comment.count 2
comment.range '0 - 31'
iface MIXER
name 'Headphone Playback Volume'
value.0 26
value.1 26
}
control.4 {
comment.access 'read write'
comment.type BOOLEAN
comment.count 2
iface MIXER
name 'Headphone Playback Switch'
value.0 true
value.1 true
}
control.5 {
comment.access 'read write'
comment.type INTEGER
comment.count 2
comment.range '0 - 31'
iface MIXER
name 'Line In Volume'
value.0 23
value.1 23
}
```

关于 amixer 命令的具体用法，读者可以参照 alsa_amixer contents 中的内容。关于编写 asound.conf 文件的方法，可以参照 alsa_alsactl 生成/etc/asound.state 的过程。下面的代码是笔者编写的 asound.conf 文件。

```
##
## Mixer Devices
##
ctl.AndroidPlayback {
type hw
card 0 # Can replace with driver"s name from /proc/asound/cards
}
ctl.AndroidRecord {
type hw
card 0 # Can replace with driver"s name from /proc/asound/cards
}
##
## Playback Devices
##
pcm.AndroidPlayback {
type hooks
slave.pcm {
type hw
card 0
device 0 # Must be of type "digital audio playback"
}
hooks.0 {
```

```
type ctl_elems
hook_args [
{
name 'Master Playback Switch'
value true
}
{
name 'Master Playback Volume'
value.0 51
value.1 51
}
{
name 'Phone Playback Switch'
value false
}
{
name 'Phone Playback Volume'
value.0 0
value.1 0
}
{
name 'Mic Playback Switch'
value false
}
{
name 'Mic Playback Volume'
value.0 0
value.1 0
}
{
name 'Mic Boost (+20dB)'
value false
}
{
name 'Line Playback Switch'
value false
}
{
name 'Line Playback Volume'
value.0 0
value.1 0
}
{
name 'PCM Playback Switch'
value true
}
{
name 'PCM Playback Volume'
value.0 51
value.1 51
```



```
}
{
name 'Capture Source'
value.0 Mic
value.1 Mic
}
{
name 'Capture Switch'
value true
}
{
name 'Capture Volume'
value.0 0
value.1 0
}
]
}
}
```

上述代码只是 asound.conf 文件的一部分，其他 pcm.AndroidPlayback_xxx 的写法都类似，唯一的区别是 hook_argsp[]中的内容需要根据自己的情况来设置。

19.6.3 在 OMAP 平台移植 Android 的 ALSA 声卡驱动

下面将讲解在 OMAP3530 平台上移植 Android 的 ALSA 声卡驱动的方法，我们以最难移植操作的 Android 5.0 为例进行讲解。

（1）使用 GIT 下载移植代码需要注意的是，不同人在网上下载的移植代码可能会不同，我们需要明确在 AudioSystem 类中是否定义了 DEVICE_OUT_EARPIECE，我们要选择使用没有定义的。

先运行如下命令：

```
git clone git://gitorious.org/android-on-freerunner/platform_external_alsa-lib.git
```

将下载的内容复制到 external 目录下，并重命名为 alsa-lib。

然后运行如下命令：

```
git clone git://gitorious.org/android-on-freerunner/platform_hardware_alsa_sound.git
```

将下载的内容复制到 hardware 目录下，并重命名为 libaudio-alsa。

再运行如下命令：

```
git clone git://gitorious.org/android-on-freerunner/platform_external_alsa-utils.git
```

将下载的内容复制到 external 目录下，并重命名为 alsa-utils。

然后通过以下命令下载 DEVICE_OUT_EARPIECE 的代码。

```
git clone git://android.git.kernel.org/platform/external/alsa-lib.git
git clone git://android.git.kernel.org/platform/external/alsa-utils.git
git clone git://android.git.kernel.org/platform/hardware/alsa_sound.git
```

复制和重命名 DEVICE_OUT_EARPIECE 的方法和前面的方法相同，不再赘述。

（2）修改文件 system/core/init/device.c，在里面加上如下代码以创建/dev/snd。

```
        } else if(!strncmp(uevent->subsystem, "mtd", 3)) {
base = "/dev/mtd/";
mkdir(base, 0755);
} else if(!strncmp(uevent->subsystem, "sound", 5)) {
```

```
base = "/dev/snd/";
mkdir(base, 0755);
```

（3）修改文件 system/core/init/devices.c 的目的是增加设备节点及权限。

```
static struct perms_ devperms[] = {
...
{ "/dev/snd/", 0664, AID_SYSTEM, AID_AUDIO, 1 },
...
```

（4）修改文件 build/target/board/generic/BoardConfig.mk。

```
# config.mk
#
# Product-specific compile-time definitions.
#

# The generic product target doesn't have any hardware-specific pieces.
TARGET_NO_BOOTLOADER := true
TARGET_NO_KERNEL := true
TARGET_NO_RADIOIMAGE := true
#HAVE_HTC_AUDIO_DRIVER := true
BOARD_USES_ALSA_AUDIO := true
BUILD_WITH_ALSA_UTILS := true
#BOARD_USES_GENERIC_AUDIO := true
BOARD_USES_GENERIC_AUDIO := false
```

（5）修改文件 hardware/alsa_sound/Android.mk，此步骤很重要，否则不会编译通过。

```
# hardware/libaudio-alsa/Android.mk
#
# Copyright 2008 Wind River Systems
#

ifeq ($(strip $(BOARD_USES_ALSA_AUDIO)),true)

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_ARM_MODE := arm
LOCAL_CFLAGS := -D_POSIX_SOURCE
    # LOCAL_WHOLE_STATIC_LIBRARIES := libasound

LOCAL_C_INCLUDES += external/alsa-lib/include

LOCAL_SRC_FILES := AudioHardwareALSA.cpp

LOCAL_MODULE := libaudio

LOCAL_STATIC_LIBRARIES += libaudiointerface \
#     libasound

LOCAL_SHARED_LIBRARIES := \
      libcutils \
      libutils \
      libmedia \
```

```
    libhardware_legacy \
    libdl \
    libc \
    libasound

include $(BUILD_SHARED_LIBRARY)

endif
```

（6）重建如下编译选项。

☑　build/envsetup.sh：不同下载的脚本名字可能有不同。

☑　choose combo：选择组合。

☑　make clean：必须经过此过程，否则不能在 Android 系统中发声。

（7）编译 make -j4//core dual。

（8）制作文件系统。

在文件/system/etc/asound.conf()中需要注意如下几个特别的配置。

```
view plaincopy to clipboardprint?
ctl.AndroidOut {
type hw
card 0
}
ctl.AndroidIn {
type hw
card 0
}
pcm.AndroidPlayback {
type hw
card 0
device 0
}
pcm.AndroidRecord {
type hw
card 0
device 0
ctl.AndroidOut {
type hw
card 0
}
ctl.AndroidIn {
type hw
card 0
}
pcm.AndroidPlayback {
type hw
card 0
device 0
}
pcm.AndroidRecord {
type hw
card 0
```

```
device 0
}
```

（9）在编译后修改文件 init.rc，重新设置 Audio 驱动设备节点的 owner 和访问属性。

```
chown root audio /dev/snd/controlC0
chown root audio /dev/snd/pcmC0D0c
chown root audio /dev/snd/pcmC0D0p
chown root audio /dev/snd/timer
chmod 0666 /dev/snd/controlC0
chmod 0666 audio /dev/snd/pcmC0D0c
chmod 0666 audio /dev/snd/pcmC0D0p
chmod 0666 audio /dev/snd/timer
```

19.6.4　基于 ARM 的 AC97 音频驱动

本节将通过一段完整的演示代码来讲解创建声卡驱动的具体流程。本实例基于 ARM 处理器的 PXA2XXAC97 芯片，其驱动程序源码位于 Linux 内核代码/sound/arm 目录下，核心实现文件是 pxa2xx-ac97.c。文件 pxa2xx-ac97.c 实现了创建和卸载声卡功能，具体实现代码如下所示。

```
static int __devinit pxa2xx_ac97_probe(struct platform_device *dev)
{
        struct snd_card *card;
        struct snd_ac97_bus *ac97_bus;
        struct snd_ac97_template ac97_template;
        int ret;
        pxa2xx_audio_ops_t *pdata = dev->dev.platform_data;

        if (dev->id >= 0) {
              dev_err(&dev->dev, "PXA2xx has only one AC97 port.\n");
              ret = -ENXIO;
              goto err_dev;
        }

        ret = snd_card_create(SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1,
                              THIS_MODULE, 0, &card);
        if (ret < 0)
              goto err;

        card->dev = &dev->dev;
        strncpy(card->driver, dev->dev.driver->name, sizeof(card->driver));

        ret = pxa2xx_pcm_new(card, &pxa2xx_ac97_pcm_client, &pxa2xx_ac97_pcm);
        if (ret)
              goto err;

        ret = pxa2xx_ac97_hw_probe(dev);
        if (ret)
              goto err;

        ret = snd_ac97_bus(card, 0, &pxa2xx_ac97_ops, NULL, &ac97_bus);
        if (ret)
```

```
            goto err_remove;
        memset(&ac97_template, 0, sizeof(ac97_template));
        ret = snd_ac97_mixer(ac97_bus, &ac97_template, &pxa2xx_ac97_ac97);
        if (ret)
            goto err_remove;

        snprintf(card->shortname, sizeof(card->shortname),
             "%s", snd_ac97_get_short_name(pxa2xx_ac97_ac97));
        snprintf(card->longname, sizeof(card->longname),
             "%s (%s)", dev->dev.driver->name, card->mixername);

        if (pdata && pdata->codec_pdata[0])
            snd_ac97_dev_add_pdata(ac97_bus->codec[0], pdata->codec_pdata[0]);
        snd_card_set_dev(card, &dev->dev);
        ret = snd_card_register(card);
        if (ret == 0) {
            platform_set_drvdata(dev, card);
            return 0;
        }

    err_remove:
        pxa2xx_ac97_hw_remove(dev);
    err:
        if (card)
            snd_card_free(card);
    err_dev:
        return ret;
    }
……
    static struct platform_driver pxa2xx_ac97_driver = {
        .probe = pxa2xx_ac97_probe,
        .remove = __devexit_p(pxa2xx_ac97_remove),
        .driver = {
            .name     = "pxa2xx-ac97",
            .owner    = THIS_MODULE,
    #ifdef CONFIG_PM
            .pm = &pxa2xx_ac97_pm_ops,
    #endif
        },
    };

    static int __init pxa2xx_ac97_init(void)
    {
        return platform_driver_register(&pxa2xx_ac97_driver);
    }

    static void __exit pxa2xx_ac97_exit(void)
    {
        platform_driver_unregister(&pxa2xx_ac97_driver);
    }

    module_init(pxa2xx_ac97_init);
```

```
module_exit(pxa2xx_ac97_exit);

MODULE_AUTHOR("Nicolas Pitre");
MODULE_DESCRIPTION("AC97 driver for the Intel PXA2xx chip");
MODULE_LICENSE("GPL");
MODULE_ALIAS("platform:pxa2xx-ac97");
```

在上述代码中，函数 pxa2xx_ac97_probe()是整个文件的核心，首先调用函数 snd_card_create()为结构体 snd_card 分配了内存空间，并设置了很多 snd_card 结构体的成员变量。函数 snd_card_create()在文件 sound/core/init.c 中定义，具体实现代码如下所示。

```
int snd_card_create(int idx, const char *xid,
            struct module *module, int extra_size,
            struct snd_card **card_ret)
{
    struct snd_card *card;
    int err, idx2;

    if (snd_BUG_ON(!card_ret))
        return -EINVAL;
    *card_ret = NULL;

    if (extra_size < 0)
        extra_size = 0;
    card = kzalloc(sizeof(*card) + extra_size, GFP_KERNEL);
    if (!card)
        return -ENOMEM;
    if (xid)
        strlcpy(card->id, xid, sizeof(card->id));
    err = 0;
    mutex_lock(&snd_card_mutex);
    if (idx < 0) {
        for (idx2 = 0; idx2 < SNDRV_CARDS; idx2++)
            /** idx == -1 == 0xffff means: take any free slot */
            if (~snd_cards_lock & idx & 1<<idx2) {
                if (module_slot_match(module, idx2)) {
                    idx = idx2;
                    break;
                }
            }
    }
    if (idx < 0) {
        for (idx2 = 0; idx2 < SNDRV_CARDS; idx2++)
            /** idx == -1 == 0xffff means: take any free slot */
            if (~snd_cards_lock & idx & 1<<idx2) {
                if (!slots[idx2] || !*slots[idx2]) {
                    idx = idx2;
                    break;
                }
            }
    }
    if (idx < 0)
```

```
			err = -ENODEV;
		else if (idx < snd_ecards_limit) {
			if (snd_cards_lock & (1 << idx))
				err = -EBUSY;	/** invalid */
		} else if (idx >= SNDRV_CARDS)
			err = -ENODEV;
		if (err < 0) {
			mutex_unlock(&snd_card_mutex);
			snd_printk(KERN_ERR "cannot find the slot for index %d (range 0-%i), error: %d\n",
				 idx, snd_ecards_limit - 1, err);
			goto __error;
		}
		snd_cards_lock |= 1 << idx;		/** lock it */
		if (idx >= snd_ecards_limit)
			snd_ecards_limit = idx + 1; /** increase the limit */
		mutex_unlock(&snd_card_mutex);
		card->number = idx;
		card->module = module;
		INIT_LIST_HEAD(&card->devices);
		init_rwsem(&card->controls_rwsem);
		rwlock_init(&card->ctl_files_rwlock);
		INIT_LIST_HEAD(&card->controls);
		INIT_LIST_HEAD(&card->ctl_files);
		spin_lock_init(&card->files_lock);
		INIT_LIST_HEAD(&card->files_list);
		init_waitqueue_head(&card->shutdown_sleep);
	#ifdef CONFIG_PM
		mutex_init(&card->power_lock);
		init_waitqueue_head(&card->power_sleep);
	#endif
		/** the control interface cannot be accessed from the user space until */
		/** snd_cards_bitmask and snd_cards are set with snd_card_register */
		err = snd_ctl_create(card);
		if (err < 0) {
			snd_printk(KERN_ERR "unable to register control minors\n");
			goto __error;
		}
		err = snd_info_card_create(card);
		if (err < 0) {
			snd_printk(KERN_ERR "unable to create card info\n");
			goto __error_ctl;
		}
		if (extra_size > 0)
			card->private_data = (char *)card + sizeof(struct snd_card);
		*card_ret = card;
		return 0;

	  __error_ctl:
		snd_device_free_all(card, SNDRV_DEV_CMD_PRE);
	  __error:
		kfree(card);
```

```
    return err;
}
EXPORT_SYMBOL(snd_card_create);
```

在上述代码中，当为结构体 snd_card 分配内存时，会根据参数 extra_size 的值多分配 extra_size 指定的内存。private_data 会指向这些内存的首地址，为芯片专用数据分配内存空间。函数 snd_card_create()执行完毕之后，将 rivate_data 转换为相应的数据结构指针。此时数据结构占用的空间需要和 extra_size 参数值一致，否则将会产生未知错误。

接下来返回到文件 pxa2xx-ac97.c，调用函数 pxa2xx_pcm_new 来初始化 private_data 和其他的成员变量。函数在文件 pxa2xx-pcm.c 中定义，具体实现代码如下所示。

```
int pxa2xx_pcm_new(struct snd_card *card, struct pxa2xx_pcm_client *client,
               struct snd_pcm **rpcm)
{
    struct snd_pcm *pcm;
    int play = client->playback_params ? 1 : 0;
    int capt = client->capture_params ? 1 : 0;
    int ret;

    ret = snd_pcm_new(card, "PXA2xx-PCM", 0, play, capt, &pcm);
    if (ret)
        goto out;

    pcm->private_data = client;
    pcm->private_free = pxa2xx_pcm_free_dma_buffers;

    if (!card->dev->dma_mask)
        card->dev->dma_mask = &pxa2xx_pcm_dmamask;
    if (!card->dev->coherent_dma_mask)
        card->dev->coherent_dma_mask = 0xffffffff;

    if (play) {
        int stream = SNDRV_PCM_STREAM_PLAYBACK;
        snd_pcm_set_ops(pcm, stream, &pxa2xx_pcm_ops);
        ret = pxa2xx_pcm_preallocate_dma_buffer(pcm, stream);
        if (ret)
            goto out;
    }
    if (capt) {
        int stream = SNDRV_PCM_STREAM_CAPTURE;
        snd_pcm_set_ops(pcm, stream, &pxa2xx_pcm_ops);
        ret = pxa2xx_pcm_preallocate_dma_buffer(pcm, stream);
        if (ret)
            goto out;
    }

    if (rpcm)
        *rpcm = pcm;
    ret = 0;

 out:
    return ret;
```

```
}

EXPORT_SYMBOL(pxa2xx_pcm_new);

MODULE_AUTHOR("Nicolas Pitre");
MODULE_DESCRIPTION("Intel PXA2xx PCM DMA module");
MODULE_LICENSE("GPL");
```

再次返回到文件 pxa2xx-ac97.c，最后调用文件 sound/core/init.c 中的函数 snd_card_register 注册声卡，创建 sysfs 系统。函数 snd_card_register()的具体实现代码如下所示。

```
int snd_card_register(struct snd_card *card)
{
	int err;

	if (snd_BUG_ON(!card))
		return -EINVAL;

	if (!card->card_dev) {
		card->card_dev = device_create(sound_class, card->dev,
					       MKDEV(0, 0), card,
					       "card%i", card->number);
		if (IS_ERR(card->card_dev))
			card->card_dev = NULL;
	}

	if ((err = snd_device_register_all(card)) < 0)
		return err;
	mutex_lock(&snd_card_mutex);
	if (snd_cards[card->number]) {
		/** already registered */
		mutex_unlock(&snd_card_mutex);
		return 0;
	}
	snd_card_set_id_no_lock(card, card->id[0] == '\0' ? NULL : card->id);
	snd_cards[card->number] = card;
	mutex_unlock(&snd_card_mutex);
	init_info_for_card(card);
#if defined(CONFIG_SND_MIXER_OSS) || defined(CONFIG_SND_MIXER_OSS_MODULE)
	if (snd_mixer_oss_notify_callback)
		snd_mixer_oss_notify_callback(card, SND_MIXER_OSS_NOTIFY_REGISTER);
#endif
	if (card->card_dev) {
		err = device_create_file(card->card_dev, &card_id_attrs);
		if (err < 0)
			return err;
		err = device_create_file(card->card_dev, &card_number_attrs);
		if (err < 0)
			return err;
	}

	return 0;
}
```

第 20 章　Overlay 系统驱动详解

在 Android 系统中，视频输出系统对应的是 Overlay 子系统，此系统是 Android 的一个可选系统，用于加速显示输出视频数据。视频输出系统的硬件通常叠加在主显示区之上的额外的叠加显示区。这个额外的叠加显示区和主显示区使用独立的显示内存。在通常情况下，主显示区用于输出图形系统，通常是 RGB 颜色空间。额外显示区用于输出视频，通常是 YUV 颜色空间。主显示区和叠加显示区通过 Blending（硬件混淆）自动显示在屏幕上。在软件部分我们无须关心叠加的实现过程，但是可以控制叠加的层次顺序和叠加层的大小等内容。本章将详细讲解 Android 视频输出系统 Overlay 驱动的基本架构知识和移植方法，为读者学习本书后面的知识打下基础。

20.1　视频输出系统结构

Overlay 系统的基本层次结构如图 20-1 所示。

Android 中的 Overlay 系统没有 Java 部分，在里面只包含了视频输出的驱动程序、硬件抽象层和本地框架等。Overlay 系统的具体结构如图 20-2 所示。

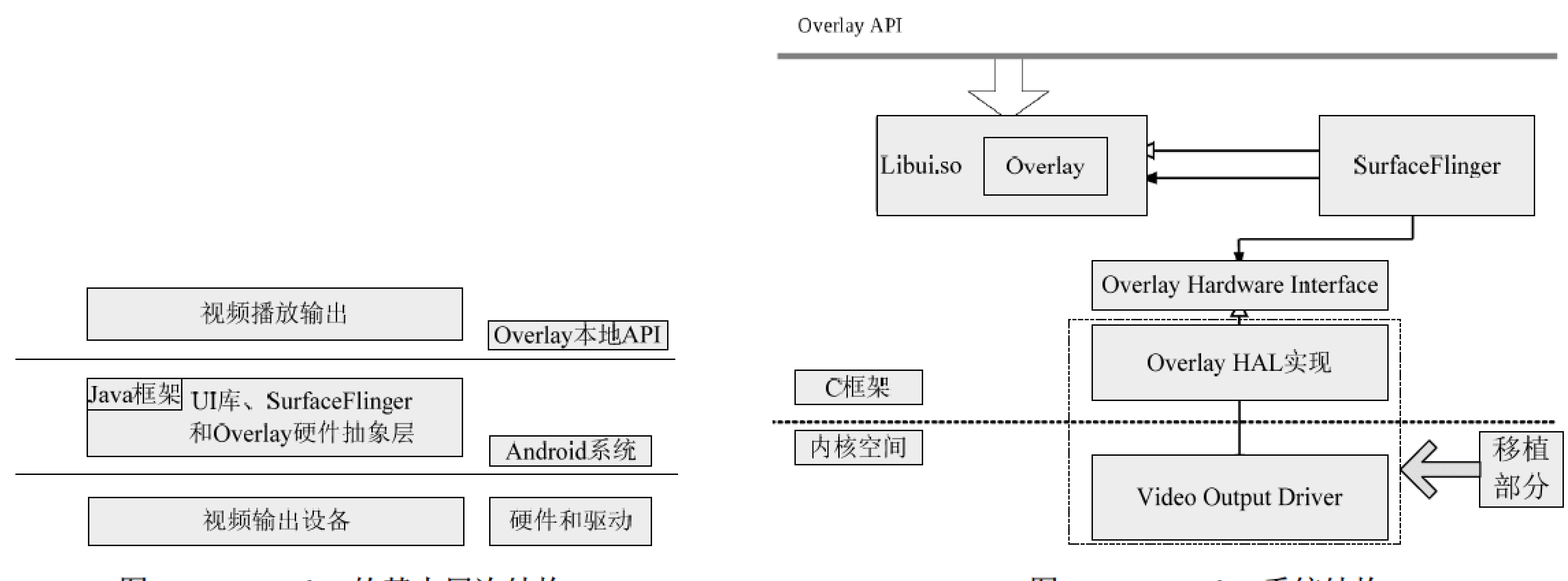


图 20-1　Overlay 的基本层次结构　　图 20-2　Overlay 系统结构

在图 20-2 所示的系统结构中，各个构成部分的具体说明如下。

（1）Overlay 驱动程序

通常是基于 FrameBuffer 或 V4L2 的驱动程序。在此文件中主要定义了两个 struct，分别是 data device 和 control device，这两个结构体分别针对 data device 和 control device 的函数 open()和函数 close()。这两个函数是注册到 device_module 中的函数。

（2）Overlay 硬件抽象层

代码路径是 hardware/qcom/display/liboverlay/overlay.h。

Overlay 硬件抽象层是一个 Android 中标准的硬件模块，其接口只有一个头文件。

（3）Overlay 服务部分

代码路径是 frameworks/native/services/surfaceflinger/。

由此可见，Overlay 系统的服务部分包含在 SurfaceFlinger 中，此层次的内容比较简单，主要功能是通过类 LayerBuffer 实现的。首先要明确的是 SurfaceFlinger 只是负责控制 merge Surface，例如计算出两个 Surface 重叠的区域，至于 Surface 需要显示的内容，则通过 Skia、Opengl 和 Pixflinger 来计算。所以在介绍 SurfaceFlinger 之前可忽略里面存储的内容是什么，先弄清楚它对 merge 的一系列控制的过程，然后再结合 2D、3D 引擎来看它的处理过程。

（4）本地框架代码

在 Overlay 系统中，本地框架的头文件路径是 frameworks/native/include/ui。

源代码路径是 frameworks/native/libs/ui。

Overlay 系统只是整个框架的一部分，主要功能是通过类 Ioverlay 和 Overlay 实现的，源代码被编译成 libui.so，它提供的 API 主要在视频输出和照相机取景模块中使用。

20.2　移植 Overlay 系统

在 Android 系统中，因为 Overlay 系统的底层和系统框架接口是硬件抽象层，所以要想实现 Overlay 系统，需要先实现硬件抽象层和下面的驱动程序。在 Overlay 系统的硬件抽象层中，使用了 Android 标准硬件模块的接口，此接口是标准 C 语言接口，通过函数和指针来实现具体功能。在里面包含了数据流接口和控制接口，我们需要根据硬件平台的具体情况来实现。

在 Android 系统中，Overlay 系统的驱动程序通常是视频输出驱动程序，可以通过标准的 FrameBuffer 驱动程序或 Video for Linux 2 视频输出驱动程序来实现。因为系统的不同，即使使用同一种驱动程序，也拥有不同的实现方式。

（1）FrameBuffer 驱动程序方式

FrameBuffer 驱动程序方式是最直接的方式，实现视频输出从驱动程序的角度和一般 FrameBuffer 驱动程序类似。区别是视频输出通过 YUV 格式颜色空间，而用于图形界面的 FrameBuffer 使用 RGB 颜色和空间。

（2）Video for Linux 2 方式

Video for Linux 2 是 Linux 视频系统的一个标准框架，在其第一个版本 Video for Linux 2 中提供了摄像头视频输入框架和视频输出接口，使用此视频输出接口，可以根据系统的性能来调整队列的数目。

20.3　硬件抽象层详解

Android 系统中的 Overlay 硬件抽象层是一个硬件模块，本节将详细讲解 Overlay 系统的硬件抽象层的基本知识，为读者学习本书后面的知识打下基础。

20.3.1　Overlay 系统硬件抽象层的接口

在 Android 系统中，在文件 hardware/qcom/display/liboverlay/overlay.h 中定义了 Overlay 系统硬件抽象层的接口，在里面主要定义了 data device 和 control device 两个 struct，并且提供了针对 data device 和 control device 的函数 open()和 close()。文件 overlay.h 的代码结构如图 20-3 所示。

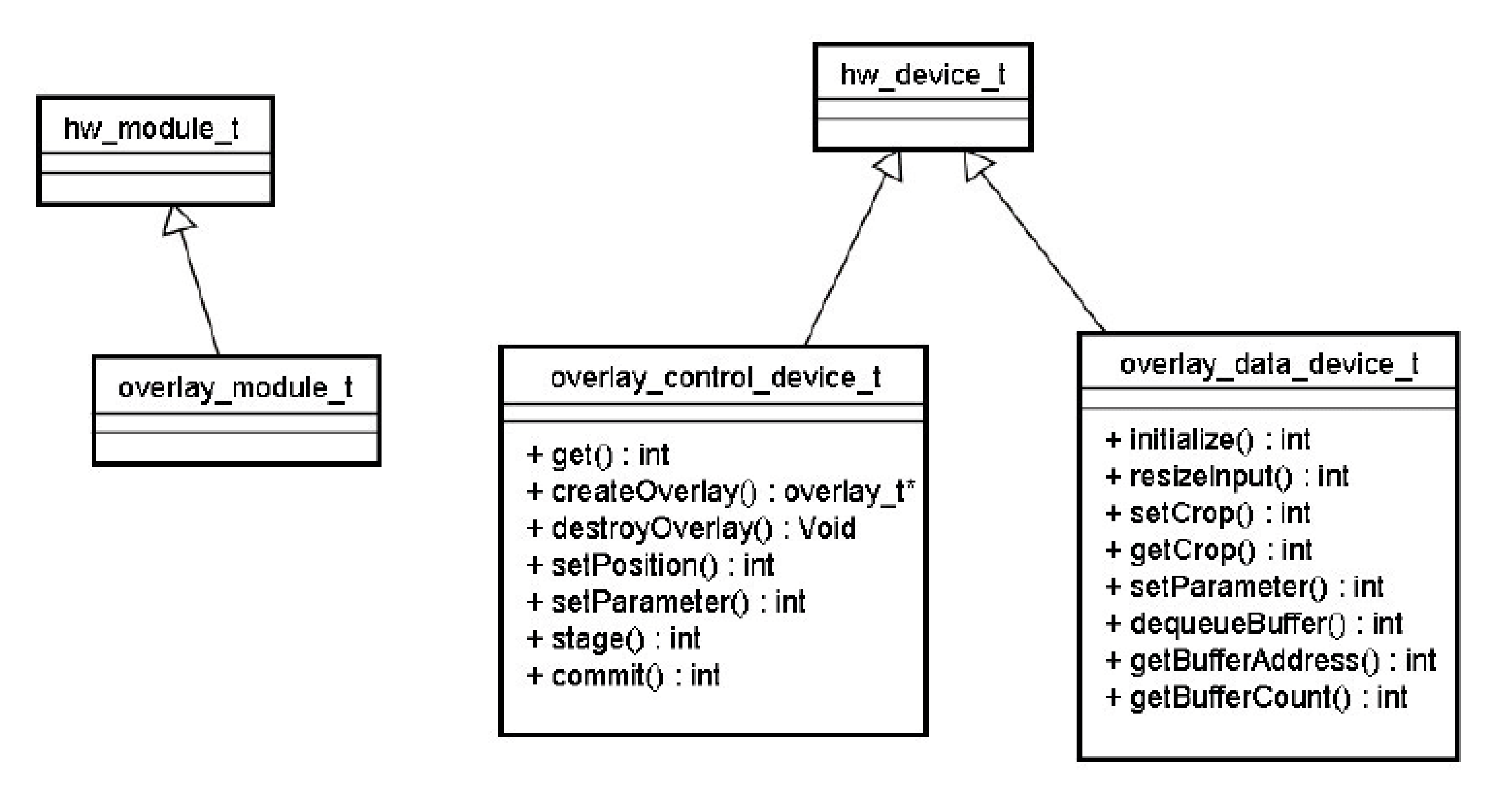


图 20-3　文件 overlay.h 结构

（1）定义 Overlay 控制设备和 Overlay 数据设备，其名称被定义为如下所示的两个字符串。

```
#define OVERLAY_HARDWARE_CONTROL      "control"
#define OVERLAY_HARDWARE_DATA         "data"
```

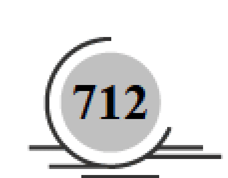
（2）在枚举 enum 中定义了所有支持的 Format，FrameBuffer 会根据 Format、width、height 来设置 Buffer（FrameBuffer 中用来显示的 Buffer）的大小。定义 enum 的具体代码如下所示。

```
enum {
    OVERLAY_FORMAT_RGBA_8888 = HAL_PIXEL_FORMAT_RGBA_8888,
    OVERLAY_FORMAT_RGB_565 = HAL_PIXEL_FORMAT_RGB_565,
    OVERLAY_FORMAT_BGRA_8888 = HAL_PIXEL_FORMAT_BGRA_8888,
    OVERLAY_FORMAT_YCbCr_422_SP = HAL_PIXEL_FORMAT_YCbCr_422_SP,
    OVERLAY_FORMAT_YCbCr_420_SP = HAL_PIXEL_FORMAT_YCbCr_420_SP,
    OVERLAY_FORMAT_YCrCb_420_SP = HAL_PIXEL_FORMAT_YCrCb_420_SP,
    OVERLAY_FORMAT_YCbYCr_422_I = HAL_PIXEL_FORMAT_YCbCr_422_I,
    OVERLAY_FORMAT_YCbYCr_420_I = HAL_PIXEL_FORMAT_YCbCr_420_I,
    OVERLAY_FORMAT_CbYCrY_422_I = HAL_PIXEL_FORMAT_CbYCrY_422_I,
    OVERLAY_FORMAT_CbYCrY_420_I = HAL_PIXEL_FORMAT_CbYCrY_420_I,
    OVERLAY_FORMAT_DEFAULT = 99      // The actual color format is determined
                                     // by the overlay
};
```

（3）定义和 Overlay 系统相关结构体，在文件 overlay.h 中，和 Overlay 系统相关结构体是 overlay_t 和 overlay_handle_t，主要实现代码如下所示。

```
typedef struct overlay_t {
    uint32_t            w;                      //宽
    uint32_t            h;                      //高
    int32_t             format;                 //颜色格式
    uint32_t            w_stride;               //一行的内容
    uint32_t            h_stride;               //一列的内容
    uint32_t            reserved[3];
    /* returns a reference to this overlay's handle (the caller doesn't
     * take ownership) */
    overlay_handle_t    (*getHandleRef)(struct overlay_t* overlay);
    uint32_t            reserved_procs[7];
} overlay_t;
```

结构体 overlay_handle_t 是在内部使用的结构体，用于保存 Overlay 硬件设备的句柄。在使用的过程中，需要从 overlay_t 获取 overlay_handle_t。其中上一层的使用只实现结构体 overlay_handle_t 指针的传递，具体



的操作是在 Overlay 的硬件抽象层中完成的。

（4）定义结构体 overlay_control_device_t，此结构体定义了一个 control device，里面的成员除了 common 都是函数，这些函数就是我们需要去实现的，在实现时会基于这个结构体扩展出一个关于 control device 的 context 的结构体，结构体 context 内部会扩充一些信息并且包含 control device。结构体 overlay_control_device_t 的具体定义代码如下所示。

```
struct overlay_control_device_t {
    struct hw_device_t common;
    int (*get)(struct overlay_control_device_t *dev, int name);
//建立设备
    overlay_t* (*createOverlay)(struct overlay_control_device_t *dev,
            uint32_t w, uint32_t h, int32_t format);
//释放资源，分配的 handle 和 control device  的内存
    void (*destroyOverlay)(struct overlay_control_device_t *dev,
            overlay_t* overlay);
//设置 overlay 的显示范围（如果是 camera 的 preview，那么 h、w 要和 preview 的 h、w 一致）
    int (*setPosition)(struct overlay_control_device_t *dev,
            overlay_t* overlay,
            int x, int y, uint32_t w, uint32_t h);
//获取 overlay 的显示范围
    int (*getPosition)(struct overlay_control_device_t *dev,
            overlay_t* overlay,
            int* x, int* y, uint32_t* w, uint32_t* h);
    int (*setParameter)(struct overlay_control_device_t *dev,
            overlay_t* overlay, int param, int value);
    int (*stage)(struct overlay_control_device_t *dev, overlay_t* overlay);
    int (*commit)(struct overlay_control_device_t *dev, overlay_t* overlay);
};
```

（5）定义结构 overlay_data_device_t，此结构和 overlay_control_device_t 类似。在具体使用上，overlay_control_device_t 实现初始化、销毁和控制类的操作，overlay_data_device_t 用于显示内存输出的数据操作。结构 overlay_data_device_t 的定义代码如下所示。

```
struct overlay_data_device_t {
    struct hw_device_t common;
//通过参数 handle  来初始化  data device
    int (*initialize)(struct overlay_data_device_t *dev,
            overlay_handle_t handle);
//重新配置显示参数 w、h。这两个参数生效需要 close 然后重新 open
    int (*resizeInput)(struct overlay_data_device_t *dev,
            uint32_t w, uint32_t h);
//下面两个分别设置显示的区域和获取显示的区域，当播放时，需要当前坐标和宽高值来定义如何显示这些数据
    int (*setCrop)(struct overlay_data_device_t *dev,
            uint32_t x, uint32_t y, uint32_t w, uint32_t h) ;
    int (*getCrop)(struct overlay_data_device_t *dev,
        uint32_t* x, uint32_t* y, uint32_t* w, uint32_t* h) ;

    int (*setParameter)(struct overlay_data_device_t *dev,
            int param, int value);

    int (*dequeueBuffer)(struct overlay_data_device_t *dev,
                overlay_buffer_t *buf);
```

```
    int (*queueBuffer)(struct overlay_data_device_t *dev,
            overlay_buffer_t buffer);
    void* (*getBufferAddress)(struct overlay_data_device_t *dev,
            overlay_buffer_t buffer);
    int (*getBufferCount)(struct overlay_data_device_t *dev);
    int (*setFd)(struct overlay_data_device_t *dev, int fd);
};
```

20.3.2 实现 Overlay 系统的硬件抽象层

在实现 Android 中 Overlay 系统的硬件抽象层时，具体实现方法取决于硬件和驱动程序，根据设备需要来进行处理，主要分为如下两种情况。

（1）FrameBuffer 驱动方式

在此方式下，需要先实现函数 getBufferAddress()，返回通过 mmap 获得 FrameBuffer 的指针即可。如果没有双缓冲的问题，不需要真正实现函数 dequeueBuffer()和 queueBuffer()。上述函数的实现文件是 overlay.cpp，此文件被保存在目录 Hardware/qcom/display/liboverlay/overlay.cpp 中。

函数 getBufferAddress()用于返回 FrameBuffer 内部显示的内存，通过 mmap 获取内存地址。函数代码如下所示。

```
void* Overlay::getBufferAddress(overlay_buffer_t buffer)
{
    if (mStatus != NO_ERROR) return NULL;
    return mOverlayData->getBufferAddress(mOverlayData, buffer);
}
```

函数 dequeueBuffer()和 queueBuffer()的实现代码如下所示。

```
status_t Overlay::dequeueBuffer(overlay_buffer_t* buffer)
{
    if (mStatus != NO_ERROR) return mStatus;
    return   mOverlayData->dequeueBuffer(mOverlayData, buffer);
}

status_t Overlay::queueBuffer(overlay_buffer_t buffer)
{
    if (mStatus != NO_ERROR) return mStatus;
    return mOverlayData->queueBuffer(mOverlayData, buffer);
}
```

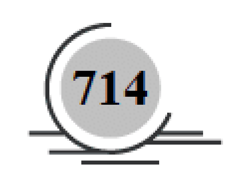
（2）Video for Linux 2 方式

如果使用 Video for Linux 2 的输出驱动，函数 dequeueBuffer()和 queueBuffer()的操作过程，和调用驱动时操作 ioctl 的主要过程是一致的，即分别调用 VIDIOC_QBUF 和 VIDIOC_DQBUF 即可直接实现。至于其他的初始化工作，可以在 initialize（初始化）中进行处理。因为存在视频数据队列，所以此处处理的内容比一般的帧缓冲区要复杂，但是可以实现更高的性能。

由此可见，在某一个硬件系统中，Overlay 的硬件层和 Overlay 系统的调用者都是特定实现的，所以只需匹配上下层代码即可实现，并不需要一一满足每一个要求，各个接口可以根据具体情况灵活使用。

20.3.3 实现 Overlay 接口

在 Android 系统中，Overlay 系统提供了 Overlay 接口，此接口用于叠加在主显示层上面的另外一个显示层。此叠加的显示层经常作为视频的输出或相机取景器的预览界面来使用。文件 Overlay.h 的主要内部实



现类是 Overlay 和 OverlayRef。OverlayRef 需要和 Surface 配合来使用，通过 ISurface 可以创建出 OverlayRef。RefBase 的主要实现代码如下所示。

```
class Overlay : public virtual RefBase
{
public:
    Overlay(const sp<OverlayRef>& overlayRef);
    void destroy();
    //获取 overlay handle，可以根据自己的需要扩展，扩展之后就有很多数据了
    overlay_handle_t getHandleRef() const;
    //获取 FrameBuffer 用于显示的内存地址
    status_t dequeueBuffer(overlay_buffer_t* buffer);
    status_t queueBuffer(overlay_buffer_t buffer);
    status_t resizeInput(uint32_t width, uint32_t height);
    status_t setCrop(uint32_t x, uint32_t y, uint32_t w, uint32_t h) ;
    status_t getCrop(uint32_t* x, uint32_t* y, uint32_t* w, uint32_t* h) ;
    status_t setParameter(int param, int value);
    void* getBufferAddress(overlay_buffer_t buffer);

    /*获取属性的信息*/
    uint32_t getWidth() const;
    uint32_t getHeight() const;
    int32_t getFormat() const;
    int32_t getWidthStride() const;
    int32_t getHeightStride() const;
    int32_t getBufferCount() const;
    status_t getStatus() const;

private:
    virtual ~Overlay();

    sp<OverlayRef> mOverlayRef;
    overlay_data_device_t *mOverlayData;
    status_t mStatus;
};

Overlay(const sp<OverlayRef>& overlayRef);
```

在上述代码中，通过 Surface 来控制 Overlay，也可以在不使用 Overlay 的情况下统一进行管理。此处是通过 OverlayRef 来创建 Overlay 的，一旦获取了 Overlay 就可以通过这个 Overlay 来获取到用来显示的 Address 地址，向 Address 中写入数据后就可以显示我们的图像了。

20.4　实现 Overlay 硬件抽象层

在本章前面的内容中，读者大致了解了 Overlay 系统的基本知识和硬件抽象层的原理。下面将详细讲解实现 Overlay 硬件抽象层的框架的基本知识。

在 Android 系统中，提供了一个 Overlay 硬件抽象层的框架实现，在里面有完整的实现代码，我们可以将其作为使用 Overlay 硬件抽象层的方法。但是因为里面没有使用具体的硬件，所以不会有实际的显示效果。Overlay 硬件抽象层框架的实现源码目录是 hardware/libhardware/modules/overlay/。

在上述目录中，主要包含了文件 Android.mk 和 overlay.cpp，其中文件 Android.mk 的主要代码如下所示。

```
LOCAL_PATH := $(call my-dir)

# HAL module implemenation, not prelinked and stored in
# hw/<OVERLAY_HARDWARE_MODULE_ID>.<ro.product.board>.so
include $(CLEAR_VARS)
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := overlay.cpp
LOCAL_MODULE := overlay.trout
include $(BUILD_SHARED_LIBRARY)
```

Overlay 库是一个 C 语言库，没有被其他库所链接，在使用时是被动打开的，所以它必须被放置在目标文件系统的 system/lib/hw 目录中。

文件 overlay.cpp 的主要代码如下所示。

```
//此结构体用于扩充 overlay_control_device_t 结构体
struct overlay_control_context_t {
    struct overlay_control_device_t device;
    /* our private state goes below here */
};
//此结构体用于扩充 overlay_data_device_t 结构体
struct overlay_data_context_t {
    struct overlay_data_device_t device;
    /* our private state goes below here */
};

//定义打开函数
static int overlay_device_open(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device);

static struct hw_module_methods_t overlay_module_methods = {
    open: overlay_device_open
};
struct overlay_module_t HAL_MODULE_INFO_SYM = {
    common: {
        tag: HARDWARE_MODULE_TAG,
        version_major: 1,
        version_minor: 0,
        id: OVERLAY_HARDWARE_MODULE_ID,
        name: "Sample Overlay module",
        author: "The Android Open Source Project",
        methods: &overlay_module_methods,
    }
static int overlay_device_open(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device)
{
    int status = -EINVAL;
    if (!strcmp(name, OVERLAY_HARDWARE_CONTROL)) {          //Overlay 的控制设备
        struct overlay_control_context_t *dev;
        dev = (overlay_control_context_t*)malloc(sizeof(*dev));

        /* initialize our state here */
```

```
        memset(dev, 0, sizeof(*dev));                              //初始化结构体

        /* initialize the procs */
        dev->device.common.tag = HARDWARE_DEVICE_TAG;
        dev->device.common.version = 0;
        dev->device.common.module = const_cast<hw_module_t*>(module);
        dev->device.common.close = overlay_control_close;

        dev->device.get = overlay_get;
        dev->device.createOverlay = overlay_createOverlay;
        dev->device.destroyOverlay = overlay_destroyOverlay;
        dev->device.setPosition = overlay_setPosition;
        dev->device.getPosition = overlay_getPosition;
        dev->device.setParameter = overlay_setParameter;

        *device = &dev->device.common;
        status = 0;
    } else if (!strcmp(name, OVERLAY_HARDWARE_DATA)) {            //Overlay 的数据设备
        struct overlay_data_context_t *dev;
        dev = (overlay_data_context_t*)malloc(sizeof(*dev));

        /* initialize our state here */
        memset(dev, 0, sizeof(*dev));                              //初始化结构体

        /* initialize the procs */
        dev->device.common.tag = HARDWARE_DEVICE_TAG;
        dev->device.common.version = 0;
        dev->device.common.module = const_cast<hw_module_t*>(module);
        dev->device.common.close = overlay_data_close;

        dev->device.initialize = overlay_initialize;
        dev->device.dequeueBuffer = overlay_dequeueBuffer;
        dev->device.queueBuffer = overlay_queueBuffer;
        dev->device.getBufferAddress = overlay_getBufferAddress;

        *device = &dev->device.common;
        status = 0;
    }
    return status;
}
```

20.5　实战演练——在 OMAP 平台实现 Overlay 系统

经过本章前面内容的学习，了解了 Overlay 系统的基本知识并分析了框架源码和 Android 源码，本节将简要介绍在 OMAP 平台中实现 Overlay 系统的基本知识。

20.5.1　实现输出视频驱动程序

在 OMAP 平台中，实现视频输出驱动程序的代码保存在目录 drivers/media/video/omap-vout/中。

在上述目录中，包含的主要文件如下。

☑ 文件 omapvout-dss.c 和 omapvout-dss.h：封装了 DSS 系统的功能。
☑ 文件 omapvout-mem.c 和 omapvout-mem.h：实现内存映射、释放和分配等功能。
☑ 文件 omapvout-vbq.c 和 omapvout-vbq.h：用于操作虚拟内存。
☑ 文件 omapvout.c 和 omapvout.h：这是 OMAP 平台的主框架，用于注册 V412 输出驱动程序的接口。

在文件 omapvout.c 中定义函数 omapvout_probe()，在此函数中建立了多个 Video 输出设备。此函数的主要实现代码如下所示。

```
static int __init omapvout_probe(struct platform_device *pdev,
                                        enum omap_plane plane, int vid)
{
        struct omapvout_device *vout = NULL;
        int rc = 0;
        DBG("omapvout_probe %d %d\n", plane, vid);
        vout = kzalloc(sizeof(struct omapvout_device), GFP_KERNEL);
        if (vout == NULL) {
                rc = -ENOMEM;
                goto err0;
        }
        mutex_init(&vout->mtx);
        vout->max_video_width = OMAPVOUT_VIDEO_MAX_WIDTH;
        vout->max_video_height = OMAPVOUT_VIDEO_MAX_HEIGHT;
        vout->max_video_bytespp = OMAPVOUT_VIDEO_MAX_BPP;
        rc = omapvout_dss_init(vout, plane);
        if (rc != 0) {
                printk(KERN_INFO "DSS init failed\n");
                goto cleanup;
        }
#ifdef CONFIG_VIDEO_OMAP_VIDEOOUT_BUFPOOL
        vout->bp = dev_get_drvdata(&pdev->dev);
        omapvout_bp_init(vout);
#endif
        /* 注册 V4L2 接口 */
        vout->vdev = omapvout_devdata;
        video_set_drvdata(&vout->vdev, vout);
        if (video_register_device(&vout->vdev, VFL_TYPE_GRABBER, vid) < 0) {
                printk(KERN_ERR MODULE_NAME": could not register with V4L2\n");
                rc = -EINVAL;
                goto cleanup;
        }
        vout->id = plane;
        return 0;
cleanup:
        omapvout_free_resources(vout);
err0:
        dev_err(&pdev->dev, "failed to setup omapvout\n");
        return rc;
}
```

在文件 omapvout.c 中定义 ideo_device 类型的结构 omapvout_devdata，此结构体是在函数 omapvout_probe_device 中被注册的 Video 设备。结构 omapvout_devdata 的定义代码如下所示。

```
static struct video_device omapvout_devdata = {
		.name = MODULE_NAME,
		.fops = &omapvout_fops,
		.ioctl_ops = &omapvout_ioctl_ops,
		.vfl_type = VID_TYPE_OVERLAY | VID_TYPE_CHROMAKEY,
		.release = video_device_release,
		.minor = -1,
};
```

20.5.2　实现 Overlay 硬件抽象层

在 OMAP 平台中，通过 Android 系统实现了 Overlay 硬件抽象层，此硬件抽象层是基于 v412 视频驱动程序实现的。OMAP 平台的 Overlay 硬件抽象层在目录 hardware/ti/omap3/liboverlay/中实现。

上述目录中的构成文件如图 20-4 所示。

Parent Directory		-
Android.mk	22-Jul-2010 17:08	1.0K
overlay.cpp	22-Jul-2010 17:08	33K
v4l2_utils.c	22-Jul-2010 17:08	16K
v4l2_utils.h	22-Jul-2010 17:08	1.9K

图 20-4　构成文件

（1）文件 Android.mk

文件 Android.mk 的主要代码如下所示。

```
LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
LOCAL_SHARED_LIBRARIES := liblog libcutils
LOCAL_SRC_FILES := v4l2_utils.c overlay.cpp
LOCAL_MODULE := overlay.omap3//设置此模块的名字是 overlay.omap3
include $(BUILD_SHARED_LIBRARY)
```

通过上述代码生成了名为 overlay.omap3 的动态库，被存放在目标系统的/system/lib/hw 中，这是 Android 标准的硬件模块。

（2）文件 overlay.cpp

在文件 overlay.cpp 中提供了 OMAP 平台的 Overlay 硬件模块框架，其中函数 overlay_createOverlay()是 Overlay 控制设备的 createOverlay()指针的实现。函数 overlay_createOverlay()的实现代码如下所示。

```
static overlay_t* overlay_createOverlay(struct overlay_control_device_t *dev,
                                         uint32_t w, uint32_t h, int32_t   format)
{
    LOGD("overlay_createOverlay:IN w=%d h=%d format=%d\n", w, h, format);
    LOG_FUNCTION_NAME;

    overlay_object              *overlay;
    overlay_control_context_t *ctx = (overlay_control_context_t *)dev;
    overlay_shared_t            *shared;

    int ret;
    uint32_t num = NUM_OVERLAY_BUFFERS_REQUESTED;
    int fd;
    int shared_fd;

    if (format == OVERLAY_FORMAT_DEFAULT)
    {
```

```
            format = OVERLAY_FORMAT_YCbYCr_422_I;
        }

        if (ctx->overlay_video1) {
            LOGE("Error - overlays already in use\n");
            return NULL;
        }

        shared_fd = create_shared_data(&shared);//创建内存共享区域
        if (shared_fd < 0) {
            LOGE("Failed to create shared data");
            return NULL;
        }
        //打开 Overlay 设备
        fd = v4l2_overlay_open(V4L2_OVERLAY_PLANE_VIDEO1);
        if (fd < 0) {
            LOGE("Failed to open overlay device\n");
            goto error;
        }初始化 Overlay 设备
        if (v4l2_overlay_init(fd, w, h, format)) {
            LOGE("Failed initializing overlays\n");
            goto error1;
        }
//设置剪切区域
        if (v4l2_overlay_set_crop(fd, 0, 0, w, h)) {
            LOGE("Failed defaulting crop window\n");
            goto error1;
        }
        //设置旋转
        if (v4l2_overlay_set_rotation(fd, 0, 0)) {
            LOGE("Failed defaulting rotation\n");
            goto error1;
        }
        //申请内存
        if (v4l2_overlay_req_buf(fd, &num, 0)) {
            LOGE("Failed requesting buffers\n");
            goto error1;
        }

      overlay = new overlay_object(fd, shared_fd, shared->size, w, h, format, num);
      if (overlay == NULL) {
            LOGE("Failed to create overlay object\n");
            goto error1;
      }
      //处理上下文
      ctx->overlay_video1 = overlay;

      overlay->setShared(shared);

      shared->controlReady = 0;
      shared->streamEn = 0;
```

```
    shared->streamingReset = 0;
    shared->dispW = LCD_WIDTH; // Need to determine this properly
    shared->dispH = LCD_HEIGHT; // Need to determine this properly

    LOGI("Opened video1/fd=%d/obj=%08lx/shm=%d/size=%d", fd,
        (unsigned long)overlay, shared_fd, shared->size);


    LOGD("overlay_createOverlay: OUT");
    return overlay;

error1:
    close(fd);
error:
    destroy_shared_data(shared_fd, shared, true);
    return NULL;
}
```

（3）文件 v4l2_utils.c

在文件 v4l2_utils.c 中，通过函数 v4l2_overlay_open()打开 Overlay 设备，此函数的实现代码如下所示。

```
int v4l2_overlay_open(int id)
{
    LOG_FUNCTION_NAME

    if (id == V4L2_OVERLAY_PLANE_VIDEO1)
        return open("/dev/video1", O_RDWR);         //打开第一个设备
    else if (id == V4L2_OVERLAY_PLANE_VIDEO2)
        return open("/dev/video2", O_RDWR);         //打开第二个设备
    return -EINVAL;
}
```

在上述代码中，参数 id 是 Overlay 设备的编号。从上述代码可以看出，在 Overlay 设备中已经包含了 dev/video 1 和 dev/video 2 两个设备。在实现硬件抽象层时，先打开第一个来解决问题，如果有需要再打开第二个。

在文件 v4l2_utils.c 中需要封装 v4l2 驱动程序，函数 v4l2_overlay_map_buf()在初始化阶段进行定义，用于从取得设备中得到内存。函数 v4l2_overlay_map_buf()的实现代码如下所示。

```
int v4l2_overlay_map_buf(int fd, int index, void **start, size_t *len)
{
    LOG_FUNCTION_NAME

    struct v4l2_buffer buf;
    int ret;
    //查询信息
    ret = v4l2_overlay_query_buffer(fd, index, &buf);
    if (ret)
        return ret;

    if (is_mmaped(&buf)) {
        LOGE("Trying to mmap buffers that are already mapped!\n");
        return -EINVAL;
    }
    *len = buf.length;
```

```
    //映射内存
    *start = mmap(NULL, buf.length, PROT_READ | PROT_WRITE, MAP_SHARED,
                        fd, buf.m.offset);
    if (*start == MAP_FAILED) {
        LOGE("map failed, length=%u offset=%u\n", buf.length, buf.m.offset);
        return -EINVAL;
    }
    return 0;
}
```

还需要调用函数 v4l2_overlay_query_buffer()来调用 v4l2 的 ioctl 命令来查询内存，此函数的实现代码如下所示。

```
int v4l2_overlay_query_buffer(int fd, int index, struct v4l2_buffer *buf)
{
    LOG_FUNCTION_NAME

    memset(buf, 0, sizeof(struct v4l2_buffer));

    buf->type = V4L2_BUF_TYPE_VIDEO_OUTPUT;          //输出 Buffer 类型 Video
    buf->memory = V4L2_MEMORY_MMAP;                  //内存类型是从内核中得到的映射内存
    buf->index = index;
    LOGI("query buffer, mem=%u type=%u index=%u\n", buf->memory, buf->type,
         buf->index);
    return v4l2_overlay_ioctl(fd, VIDIOC_QUERYBUF, buf, "querybuf ioctl");
}
```

还需要定义函数 v4l2_overlay_stream_on()来打开数据流，定义函数 v4l2_overlay_stream_off()关闭数据流。这两个函数的实现代码如下所示。

```
int v4l2_overlay_stream_on(int fd)
{
    LOG_FUNCTION_NAME
    int ret;
    uint32_t type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
    ret = v4l2_overlay_set_local_alpha(fd, 1);
    if (ret)
        return ret;
    ret = v4l2_overlay_ioctl(fd, VIDIOC_STREAMON, &type, "stream on");
    return ret;
}

int v4l2_overlay_stream_off(int fd)
{
    LOG_FUNCTION_NAME
    int ret;
    uint32_t type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
    ret = v4l2_overlay_set_local_alpha(fd, 0);
    if (ret)
        return ret;
    ret = v4l2_overlay_ioctl(fd, VIDIOC_STREAMOFF, &type, "stream off");
    return ret;
}
```

还需要定义函数 v4l2_overlay_q_buf()来对应 Overlay 数据设备的 queueBuffer 接口，定义函数 v4l2_

overlay_dq_buf()来对应 Overlay 数据设备的 dequeueBuffer 接口。这两个函数的实现代码如下所示。

```
int v4l2_overlay_q_buf(int fd, int index)
{
    struct v4l2_buffer buf;
    int ret;

    /*
    ret = v4l2_overlay_query_buffer(fd, buffer_cookie, index, &buf);
    if (ret)
        return ret;
    if (is_queued(buf)) {
        LOGE("Trying to queue buffer to kernel that is already queued!\n");
        return -EINVAL
    }
    //输出 Buffer 类型 Video
    buf.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
    buf.index = index;
    //内存类型是从内核中得到的映射内存
buf.memory = V4L2_MEMORY_MMAP;
    buf.field = V4L2_FIELD_NONE;
    buf.timestamp.tv_sec = 0;
    buf.timestamp.tv_usec = 0;
    buf.flags = 0;

    return v4l2_overlay_ioctl(fd, VIDIOC_QBUF, &buf, "qbuf");
}

int v4l2_overlay_dq_buf(int fd, int *index)
{
    struct v4l2_buffer buf;
    int ret;

    /*
    ret = v4l2_overlay_query_buffer(fd, buffer_cookie, index, &buf);
    if (ret)
        return ret;

    if (is_dequeued(buf)) {
        LOGE("Trying to dequeue buffer that is not in kernel!\n");
        return -EINVAL
    }
    //输出 Buffer 类型 Video
    buf.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
    //内存类型是从内核中得到的映射内存
buf.memory = V4L2_MEMORY_MMAP;
    ret = v4l2_overlay_ioctl(fd, VIDIOC_DQBUF, &buf, "dqbuf");
    if (ret)
      return ret;
    *index = buf.index;
    return 0;
}
```

20.6 实战演练——在系统层调用 Overlay HAL

经过本章前面内容的学习，读者大致了解了 Android 系统中 Overlay 模块的基本知识，本节将详细讲解在系统层调用 Overlay HAL 架构的基本知识。

20.6.1 测试文件

在 Android 系统中，在文件 frameworks/base/libs/surfaceflinger/tests/overlays/overlays.cpp 中提供了简单调用 Overlay 的方法。

遗憾的是上述测试程序有错误，在编译时提示编译如下代码失败。

```
sp<Surface> surface = client->createSurface(getpid(), 0, 320, 240, PIXEL_FORMAT_UNKNOWN, ISurfaceComposer::
ePushBuffers);
```

其实读者朋友们无须担心，造成上述错误的原因是申请 Surface 接口失败，和 Overlay 系统无关。

```
int main(int argc, char** argv)
{
    //建立线程池
    sp<ProcessState> proc(ProcessState::self());
    ProcessState::self()->startThreadPool();

    //创建一个 SurfaceFlinger client
    sp<SurfaceComposerClient> client = new SurfaceComposerClient();

    //创建一个 Surface 界面
    sp<Surface> surface = client->createSurface(getpid(), 0, 320, 240,
            PIXEL_FORMAT_UNKNOWN, ISurfaceComposer::ePushBuffers);

    //取得 ISurface 接口
    sp<ISurface> isurface = Test::getISurface(surface);
    printf("isurface = %p\n", isurface.get());

    //创建一个 Overlay
    sp<OverlayRef> ref = isurface->createOverlay(320, 240, PIXEL_FORMAT_RGB_565);
    sp<Overlay> overlay = new Overlay(ref);
      /*
     *创建好 Overlay 后，即可使用 Overlay 的 API，这些都对应到 Overlay HAL 的具体实现
     */
      overlay_buffer_t buffer;
    overlay->dequeueBuffer(&buffer);
    printf("buffer = %p\n", buffer);
      void* address = overlay->getBufferAddress(buffer);
    printf("address = %p\n", address);
    overlay->queueBuffer(buffer);//最重要的操作就是通过 queueBuffer 将 Buffer 列队
    return 0;
}
```

20.6.2　在 Android 系统中创建 Overlay

Overlay 系统是一个功能强大的系统，不仅可实现简单的视频输出，而且还可实现与摄像头、GPS 等有关的功能。Overlay 的具体应用主要体现在如下几个方面。

（1）摄像头应用文件 CameraService.cpp（frameworks/base/camera/libcameraservice），实现流程如下所示。

```
setPreviewDisplay()、startPreviewMode()
|
setOverlay()
|
creatOverlay()
```

（2）界面相关应用文件 ISurface.cpp（frameworks/base/libs/ui）

函数 LayerBaseClient::Surface::onTransact()在文件 LayerBase.cpp 中实现，能够通过 ibind 进程实现通信功能。其中函数 BnSurface::onTransact()有 5 种方式，只有确定有 Overlay 硬件支持时才会调用如下 case CREATE_OVERLAY 语句。

```
...
switch(code) {
        case REQUEST_BUFFER: {
            CHECK_INTERFACE(ISurface, data, reply);
            int bufferIdx = data.readInt32();
            int usage = data.readInt32();
            sp<GraphicBuffer> buffer(requestBuffer(bufferIdx, usage));
            return GraphicBuffer::writeToParcel(reply, buffer.get());
        }
        case REGISTER_BUFFERS: {
            CHECK_INTERFACE(ISurface, data, reply);
            BufferHeap buffer;
            buffer.w = data.readInt32();
            buffer.h = data.readInt32();
            buffer.hor_stride = data.readInt32();
            buffer.ver_stride= data.readInt32();
            buffer.format = data.readInt32();
            buffer.transform = data.readInt32();
            buffer.flags = data.readInt32();
            buffer.heap = interface_cast<IMemoryHeap>(data.readStrongBinder());
            status_t err = registerBuffers(buffer);
            reply->writeInt32(err);
            return NO_ERROR;
        } break;
        case UNREGISTER_BUFFERS: {
            CHECK_INTERFACE(ISurface, data, reply);
            unregisterBuffers();
            return NO_ERROR;
        } break;
        case POST_BUFFER: {
            CHECK_INTERFACE(ISurface, data, reply);
            ssize_t offset = data.readInt32();
            postBuffer(offset);
            return NO_ERROR;
```

```
            } break;
            case CREATE_OVERLAY: {
                CHECK_INTERFACE(ISurface, data, reply);
                int w = data.readInt32();
                int h = data.readInt32();
                int f = data.readInt32();
                sp<OverlayRef> o = createOverlay(w, h, f);
                return OverlayRef::writeToParcel(reply, o);
            } break;
            default:
                return BBinder::onTransact(code, data, reply, flags);
...
```

（3）文件 LayerBuffer.cpp（frameworks/base/libs/surfaceflinger）是 createOverlay 的实现，具体实现代码如下所示。

```
sp<OverlayRef> LayerBuffer::SurfaceLayerBuffer::createOverlay(uint32_t w, uint32_t h, int32_t format)
|
sp<OverlayRef> LayerBuffer::createOverlay(uint32_t w, uint32_t h, int32_t f)
|
//通过 OverlaySource 创建 overlay
sp<OverlaySource> source = new OverlaySource(*this, &result, w, h, f);

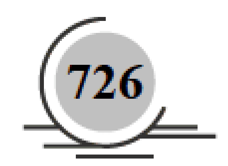
LayerBuffer::OverlaySource::OverlaySource() //此函数调用了 Overlay HAL 的 API createOverlay
{
    overlay_control_device_t* overlay_dev = mLayer.mFlinger->getOverlayEngine();//get HAL
    overlay_t* overlay = overlay_dev->createOverlay(overlay_dev, w, h, format);//HAL API
    overlay_dev->setParameter(overlay_dev, overlay, OVERLAY_DITHER, OVERLAY_ENABLE);
//设置参数，初始化 OverlayRef 类，OverlayRef 的构造函数在 Overlay.cpp 中
    mOverlay = overlay;
    mWidth = overlay->w;
    mHeight = overlay->h;
    mFormat = overlay->format;
    mWidthStride = overlay->w_stride;
    mHeightStride = overlay->h_stride;
    mInitialized = false;
...
    *overlayRef = new OverlayRef(mOverlayHandle, channel,mWidth, mHeight, mFormat, mWidthStride,
mHeightStride);
}
```

20.6.3 管理 Overlay HAL 模块

文件 Overlay.cpp（frameworks/base/libs/ui）用于负责管理 Overlay HAL 模块，并封装 HAL 的 API。具体实现流程如下。

（1）打开 Overlay HAL 模块，具体实现代码如下所示。

```
Overlay::Overlay(const sp<OverlayRef>& overlayRef)
    : mOverlayRef(overlayRef), mOverlayData(0), mStatus(NO_INIT)
{
    mOverlayData = NULL;
    hw_module_t const* module;
    if (overlayRef != 0) {
```



```
            if (hw_get_module(OVERLAY_HARDWARE_MODULE_ID, &module) == 0) {
                if (overlay_data_open(module, &mOverlayData) == NO_ERROR) {
                    mStatus = mOverlayData->initialize(mOverlayData,
                            overlayRef->mOverlayHandle);
                }
            }
        }
}
```

（2）初始化 Overlay HAL，主要实现代码如下所示。

```
overlayRef = new OverlayRef(mOverlayHandle, channel,mWidth, mHeight, mFormat, mWidthStride, mHeightStride);
```

其构造函数位于文件 Overlay.cpp 中，对应代码如下所示。

```
OverlayRef::OverlayRef(overlay_handle_t handle, const sp<IOverlay>& channel,
        uint32_t w, uint32_t h, uint32_t f, uint32_t ws, uint32_t hs)
    : mOverlayHandle(handle), mOverlayChannel(channel),
    mWidth(w), mHeight(h), mFormat(f), mWidthStride(ws), mHeightStride(hs),
    mOwnHandle(false)
{
}
```

（3）需要封装了很多需要的 API，例如 TI 自己编写的函数 opencore()用于负责视频输出。各个 API 的实现和编码请读者参考开源文件，在此不再一一讲解。

由此可以看出，虽然 Overlay 的输出对象有两种，一种是视频（主要是 YUV 格式，调用系统的 V4L2），另外一种是 ISurface 的一些图像数据（RGB 格式，直接写 FrameBuffer）。从代码实现角度看，目前 Android 系统默认并没有使用 Overlay 功能，虽然提供了 Skeleton 的 Overlay HAL 并对其进行封装，但是上层几乎没有调用到封装的 API。

如果要用好 Overlay HAL，需要大量修改上层框架，这对视屏播放可能比较重要，具体可以参考 TI 编写的实现文件 Android_surface_output_omap34xx.cpp，并且 Surface 实现的 Overlay 功能和 Copybit 的功能有部分重复，从 TI 的代码看主要是实现 V4L2 的 Overlay 功能。

20.6.4　S3C6410 Android Overlay 的测试代码

笔者用 S3C6410 的板子测试了其提供的开源代码，具体测试代码如下所示。

```
#include <binder/IPCThreadState.h>
#include <binder/ProcessState.h>
#include <binder/IServiceManager.h>
#include <utils/Log.h>
#include <ui/Surface.h>
#include <ui/ISurface.h>
#include <ui/Overlay.h>
#include <ui/SurfaceComposerClient.h>
#define FILE2 "/data/22.bin"

using namespace android;
namespace android {
class Test {
public:
    static const sp<ISurface>& getISurface(const sp<Surface>& s) {
        return s->getISurface();
    }
```

```
};
};
int main(int argc, char** argv)
{
int err;
FILE* fd;
fd = fopen(FILE2,"r");
if(fd < 0 )
{
  printf("open file err!");
  return -1;
}
    sp<ProcessState> proc(ProcessState::self());
    ProcessState::self()->startThreadPool();
    sp<SurfaceComposerClient> client = new SurfaceComposerClient();
    sp<SurfaceControl> sc = client->createSurface(getpid(), 0, 200, 200, //Surface - >SurfaceControl
            PIXEL_FORMAT_UNKNOWN, ISurfaceComposer::ePushBuffers);
  sp<Surface> surface = sc->getSurface();
    sp<ISurface> isurface = Test::getISurface(surface);
    printf("isurface = %p\n", isurface.get());
    sp<OverlayRef> ref = isurface->createOverlay(200, 200, PIXEL_FORMAT_RGB_565);
    sp<Overlay> overlay = new Overlay(ref);
  overlay->setCrop(200,50,200,200);
    overlay_buffer_t buffer;
    err = overlay->dequeueBuffer(&buffer);
    printf("buffer = %p err is %d\n", buffer,err);

    void* address = overlay->getBufferAddress(buffer);
    printf("address = %p\n", address);
  err = fread(address,1,200*200*2,fd);
if(err < 0)
  printf("read file error err is %d\n",err);
    err = overlay->queueBuffer(buffer);
  printf("queueBuffer err is %d\n",err);
  sleep(10);
    return 0;
}
```

以上代码在笔者的开发板上运行很正常，效果如图 20-5 所示。

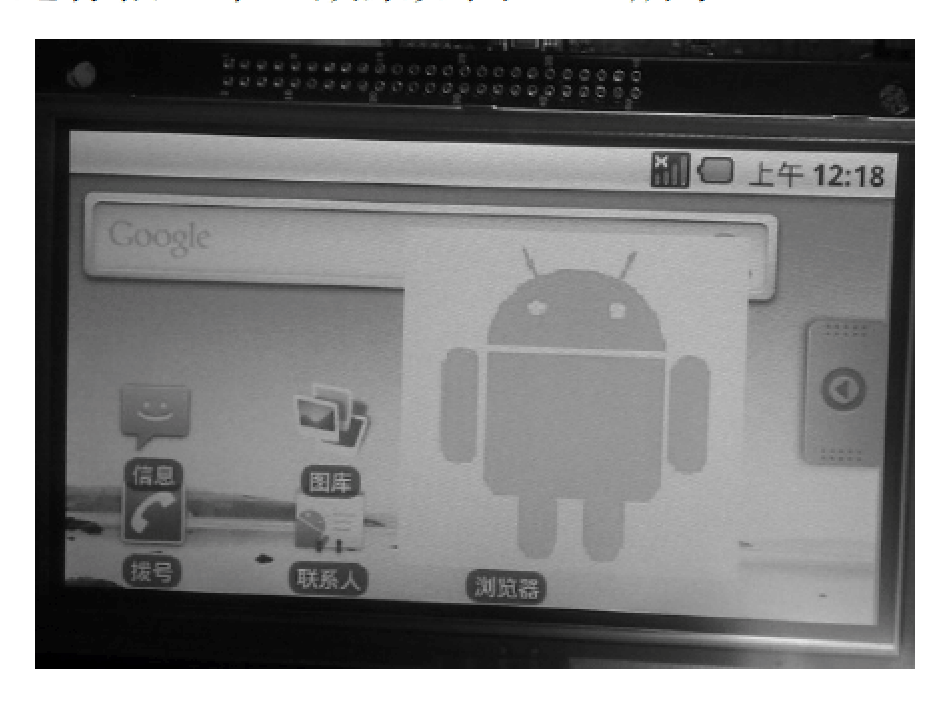


图 20-5　测试效果

第 21 章　照相机驱动

在 Android 系统中，照相机功能是通过 Camera 系统实现的。Camera 照相机系统提供了取景器、视频录制和拍摄相片等功能，并且还具有各种控制类的接口，另外还提供了 Java 层的接口和本地接口。其中 Java 框架中的 Camera 类实现了 Java 层相机接口，为照相机类和扫描类使用。而 Camera 的本地接口可以给本地程序调用，作为视频输入环节应用于摄像机和视频电话领域。本章将详细讲解 Android 平台中 Camera 系统的基本架构知识和驱动移植方法。

21.1　Camera 系统的结构

Android 照相机系统的基本层次结构如图 21-1 所示。

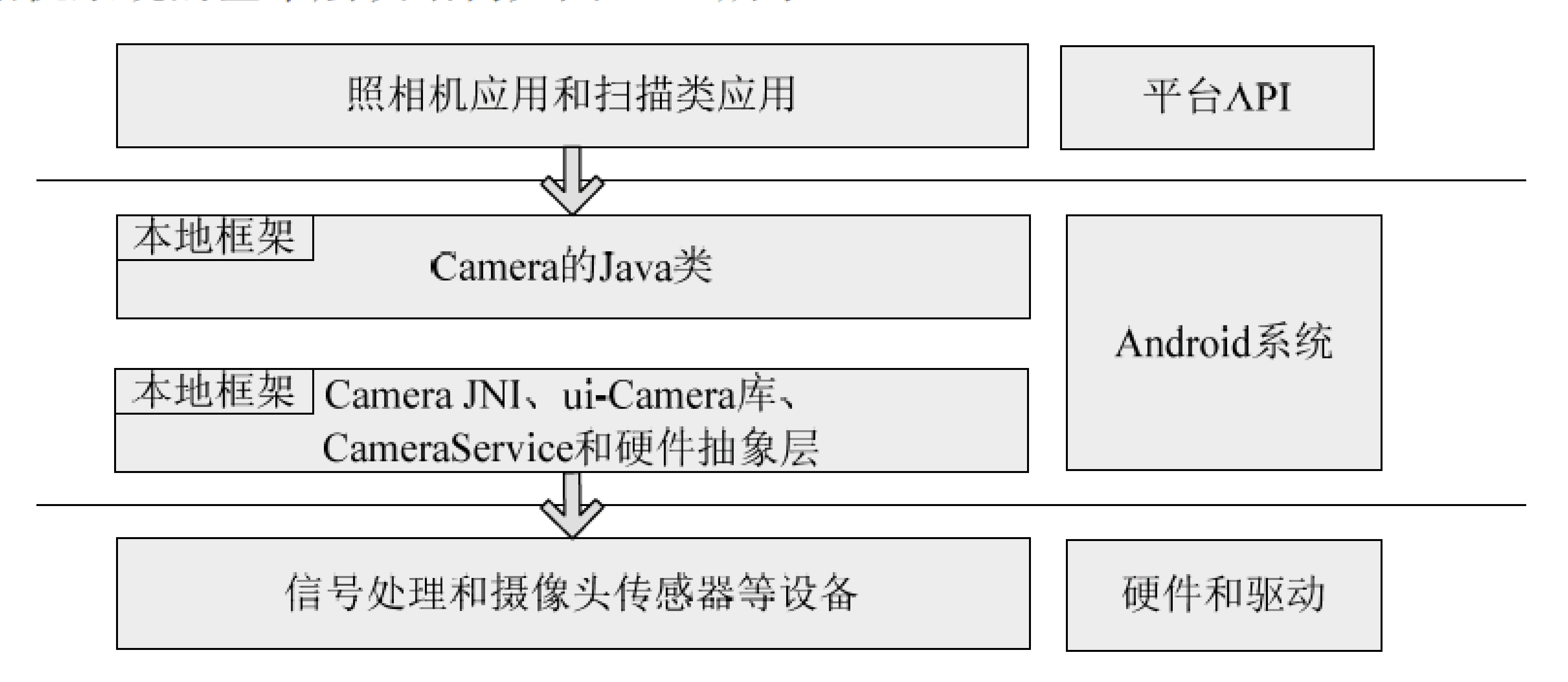


图 21-1　照相机系统的层次结构

Android 系统中的 Camera 系统包括 Camera 驱动程序层、Camera 硬件抽象层、AudioService、Camera 本地库、Camera 的 Java 框架类和 Java 应用层对 Camera 系统的调用。Camera 系统的具体结构如图 21-2 所示。

图 21-2 中各个构成层次的具体说明如下。

（1）Camera 系统的 Java 层

代码路径是 frameworks/base/core/java/android/hardware/。

其中文件 Camera.java 是主要实现的文件，对应的 Java 层次的类是 android.hardware.Camera，这个类和 JNI 中定义的是一个类，有些方法通过 JNI 的方式调用本地代码得到，有些方法自己实现。

（2）Camera 系统的 Java 本地调用部分（JNI）

代码路径是 frameworks/base/core/jni/android_hardware_Camera.cpp。

这部分内容编译成为目标文件 libandroid_runtime.so，主要的头文件在目录 frameworks/base/include/ui/中。

（3）Camera 本地框架

其中头文件路径是 frameworks/native/include/ui 或 frameworks/av/include/camera/。

源代码路径是 frameworks/native/libs/ui 或 frameworks/av/camera/。

这部分的内容被编译成库 libui.so 或 libcamera_client.so。

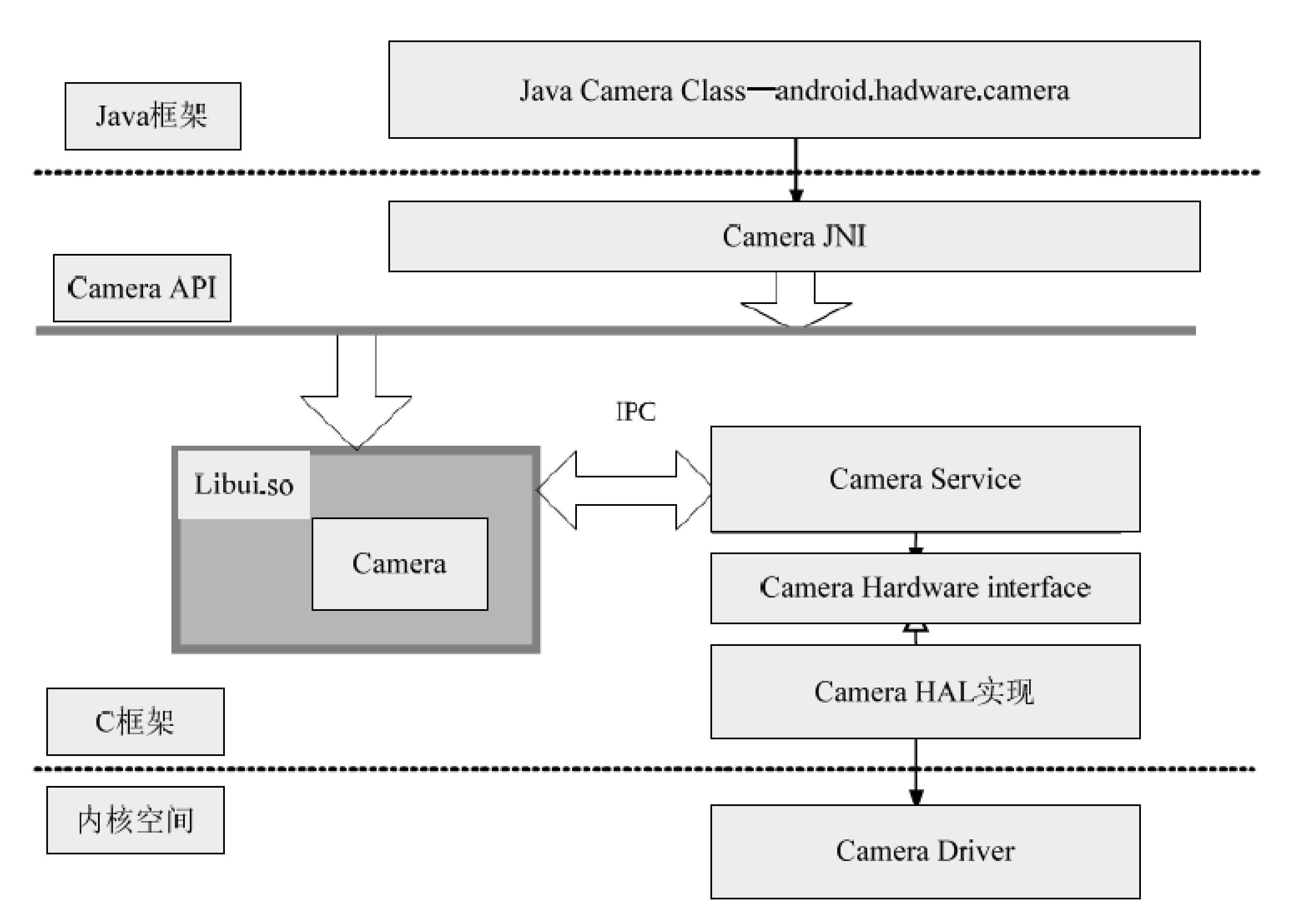


图 21-2　Camera 的系统结构

（4）Camera 服务部分

代码路径是 frameworks/av/services/camera/libcameraservice/。

这部分内容被编译成库 libcameraservice.so。

为了实现一个具体功能的 Camera 驱动程序，在最底层还需要一个硬件相关的 Camer 库（例如通过调用 video for linux 驱动程序和 Jpeg 编码程序实现）。这个库将被 Camera 的服务库 libcameraservice.so 调用。

（5）摄像头驱动程序

此部分是基于 Linux 的 Video for Linux 视频驱动框架。

（6）硬件抽象层

硬件抽象层中的接口代码路径是 frameworks/base/include/ui/或 frameworks/av/include/camera/。

其中的核心文件是 CameraHardwareInterface.h。

在 Camera 系统的各个库中，库 libui.so 位于核心的位置，它对上层提供的接口主要是 Camera 类，类 libandroid_runtime.so 通过调用 Camera 类提供对 Java 的接口，并且实现了 android.hardware.camera 类。

库 libcameraservice.so 是 Camera 的服务器程序，它通过继承 libui.so 的类实现服务器的功能，并且与 libui.so 中的另外一部分内容通过进程间通信（即 Binder 机制）的方式进行通信。

库 libandroid_runtime.so 和 libui.so 是公用库，在里面除了 Camera 外还有其他方面的功能。

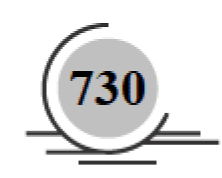
Camera 部分的头文件被保存在 frameworks/base/include/ui/目录下，此目录和库 libmedia.so 的源文件目录 frameworks/base/libs/ui/相对应。

在 Camera 中主要包含如下头文件。

☑ ICameraClient.h
☑ Camera.h
☑ ICamera.h
☑ ICameraService.h
☑ CameraHardwareInterface.h

文件 Camera.h 提供了对上层的接口，而其他的几个头文件都是提供一些接口类（即包含了纯虚函数的类），这些接口类必须被实现类继承才能够使用。

当整个 Camera 在运行时，可以大致上分成 Client 和 Server 两个部分，分别在两个进程中运行，它们之



间使用 Binder 机制实现进程间通信。这样在客户端调用接口，功能则在服务器中实现，但是在客户端中调用就好像直接调用服务器中的功能，进程间通信的部分对上层程序不可见。

从框架结构上来看，文件 ICameraService.h、ICameraClient.h 和 ICamera.h 中的 3 个类定义了 Camera 的接口和架构，ICameraService.cpp 和 Camera.cpp 两个文件用于实现 Camera 架构，Camera 的具体功能在下层调用硬件相关的接口中实现。

21.1.1　Java 程序部分

在 Android 系统中已经实现了一个 Camera 硬件抽象层的"桩"，这样可以根据"宏"来配置。此"桩"使用假的方式实现取景器预览和照片拍摄功能。在 Camera 系统的"桩"实现中使用黑白格子代替来自硬件的视频流，这样可以在不接触硬件的情况下让 Camera 系统不用硬件也可以运行。因为没有视频输出设备，所以不会使用 Overlay 来实现 Camera 硬件抽象层的"桩"。

在文件 packages/apps/Camera/src/com/android/camera/Camera.java 中，已经包含了对 Camera 的调用。在文件 Camera.java 中包含的对包的引用代码如下。

```
import android.hardware.Camera.PictureCallback;
import android.hardware.Camera.Size;
```

然后定义类 Camera，此类继承了活动 Activity 类，在它的内部包含了一个 android.hardware.Camera。对应代码如下。

```
public class Camera extends Activity implements View.OnClickListener, SurfaceHolder.Callback{
android.hardware.Camera mCameraDevice;
}
```

调用 Camera 功能的代码如下。

```
mCameraDevice.takePicture(mShutterCallback, mRawPictureCallback, mJpegPictureCallback);
mCameraDevice.startPreview();
mCameraDevice.stopPreview();
startPreview、stopPreview 和 takePicture 等接口就是通过 Java 本地调用（JNI）来实现的
frameworks/base/core/java/android/hardware/目录中的 Camera.java 文件提供了一个 Java 类：Camera
public class Camera {
}
```

在类 Camera 中，大部分代码使用 JNI 调用下层得到，例如下面的代码。

```
public void setParameters(Parameters params) {
        Log.e(TAG, "setParameters()");
        //params.dump();
        native_setParameters(params.flatten());
}
```

还有下面的代码：

```
public final void setPreviewDisplay(SurfaceHolder holder) {
        setPreviewDisplay(holder.getSurface());
}
private native final void setPreviewDisplay(Surface surface);
```

在上面的两段代码中，两个 setPreviewDisplay 参数不同，后一个是本地方法，参数为 Surface 类型，前一个通过调用后一个实现，但自己的参数以 SurfaceHolder 为类型。

21.1.2　Camera 的 Java 本地调用部分

在 Android 系统中，Camera 驱动的 Java 本地调用（JNI）部分在文件 frameworks/base/core/jni/android_

hardware_Camera.cpp 中实现。

在文件 android_hardware_Camera.cpp 中定义了一个 JNINativeMethod（Java 本地调用方法）类型的数组 gMethods，具体代码如下。

```
static JNINativeMethod camMethods[] = {
{"native_setup","(Ljava/lang/Object;)V",(void*)android_hardware_Camera_native_setup },
    {"native_release","()V",(void*)android_hardware_Camera_release },
    {"setPreviewDisplay","(Landroid/view/Surface;)V",(void *)android_hardware_Camera_setPreviewDisplay },
    {"startPreview","()V",(void *)android_hardware_Camera_startPreview },
    {"stopPreview", "()V", (void *)android_hardware_Camera_stopPreview },
    {"setHasPreviewCallback","(Z)V",(void *)android_hardware_Camera_setHasPreviewCallback },
    {"native_autoFocus","()V",(void *)android_hardware_Camera_autoFocus },
    {"native_takePicture", "()V", (void *)android_hardware_Camera_takePicture },
    {"native_setParameters","(Ljava/lang/String;)V",(void *)android_hardware_Camera_setParameters },
    {"native_getParameters", "()Ljava/lang/String;",(void *)android_hardware_Camera_getParameters }
};
```

JNINativeMethod 的第 1 个成员是一个字符串，表示 Java 本地调用方法的名称，此名称是在 Java 程序中调用的名称；第 2 个成员也是一个字符串，表示 Java 本地调用方法的参数和返回值；第 3 个成员是 Java 本地调用方法对应的 C 语言函数。

通过函数 register_android_hardware_Camera()将 gMethods 注册为类 android/media/Camera，其主要实现如下。

```
int register_android_hardware_Camera(JNIEnv *env)
{
// Register native functions
return AndroidRuntime::registerNativeMethods(env, "android/hardware/Camera",camMethods, NELEM(camMethods));
}
```

其中类 android/hardware/Camera 和 Java 类 android.hardware.Camera 相对应。

21.1.3　Camera 的本地库 libui.so

文件 frameworks/base/libs/ui/Camera.cpp 用于实现文件 Camera.h 中提供的接口，其中最重要的代码片段如下。

```
sp<Camera> Camera::create(const sp<ICamera>& camera)
{
      ALOGV("create");
      if (camera == 0) {
           ALOGE("camera remote is a NULL pointer");
           return 0;
      }

     sp<Camera> c = new Camera(-1);
     if (camera->connect(c) == NO_ERROR) {
          c->mStatus = NO_ERROR;
          c->mCamera = camera;
          camera->asBinder()->linkToDeath(c);
          return c;
     }
     return 0;
}
```

函数 connect()的实现代码如下。

```
sp<Camera> Camera::connect(int cameraId, const String16& clientPackageName,
            int clientUid)
{
    return CameraBaseT::connect(cameraId, clientPackageName, clientUid);
}
```

函数 connect()通过调用 getCameraService 得到一个 ICameraService，再通过 ICameraService 的 cs->connect(c)得到一个 ICamera 类型的指针。调用 connect()函数会得到一个 Camera 类型的指针。在正常情况下，已经初始化完成了 Camera 的成员 mCamera。

函数 startPreview()的实现代码如下。

```
status_t Camera::startPreview()
{
    ALOGV("startPreview");
    sp <ICamera> c = mCamera;
    if (c == 0) return NO_INIT;
    return c->startPreview();
}
```

其他函数的实现过程与函数 setDataSource 类似。在库 libmedia.so 中的其他文件与头文件的名称相同，分别如下。

- ☑ frameworks/base/libs/ui/ICameraClient.cpp
- ☑ frameworks/base/libs/ui/ICamera.cpp
- ☑ frameworks/base/libs/ui/ICameraService.cpp

此处的类 BnCameraClient 和 BnCameraService 虽然实现了 onTransact()函数，但是由于还有纯虚函数没有实现，所以不能实例化这个类。

21.1.4　Camera 服务 libcameraservice.so

目录 frameworks/av/services/camera/libcameraservice/实现一个 Camera 的服务，此服务是继承 ICameraService 的具体实现。在此目录下和硬件抽象层“桩”实现相关的文件说明如下。

- ☑ CameraHardwareStub.cpp：Camera 硬件抽象层“桩”实现。
- ☑ CameraHardwareStub.h：Camera 硬件抽象层“桩”实现的接口。
- ☑ CannedJpeg.h：包含一块 JPEG 数据，在拍照片时作为 JPEG 数据。
- ☑ FakeCamera.h 和 FakeCamera.cpp：实现假的 Camera 黑白格取景器效果。

在文件 Android.mk 中，使用宏 USE_CAMERA_STUB 决定是否使用真的 Camera，如果宏为真，则使用 CameraHardwareStub.cpp 和 FakeCamera.cpp 构造一个假的 Camera；如果为假，则使用 CameraService.cpp 构造一个实际的 Camera 服务。文件 Android.mk 的主要代码如下。

```
LOCAL_MODULE:= libcamerastub
LOCAL_SHARED_LIBRARIES:= libui
include $(BUILD_STATIC_LIBRARY)
endif # USE_CAMERA_STUB
#
# libcameraservice
#

include $(CLEAR_VARS)
LOCAL_SRC_FILES:=                    \
```

```
        CameraService.cpp
LOCAL_SHARED_LIBRARIES:= \
        libui \
        libutils \
        libcutils \
        libmedia
LOCAL_MODULE:= libcameraservice
LOCAL_CFLAGS+=-DLOG_TAG=\"CameraService\"
ifeq ($(USE_CAMERA_STUB), true)
LOCAL_STATIC_LIBRARIES += libcamerastub
LOCAL_CFLAGS += -include CameraHardwareStub.h
else
LOCAL_SHARED_LIBRARIES += libcamera
endif
include $(BUILD_SHARED_LIBRARY)
```

文件 CameraService.cpp 继承了 BnCameraService 的实现，在此类内部又定义了类 Client，CameraService::Client 继承了 BnCamera。在运作的过程中，函数 CameraService::connect()用于得到一个 CameraService::Client。在使用过程中，主要是通过调用这个类的接口来实现完成 Camera 的功能。因为 CameraService::Client 本身继承了 BnCamera 类，而 BnCamera 类继承了 ICamera，所以可以将此类当成 ICamera 来使用。

类 CameraService 和 CameraService::Client 的结果如下。

```
class CameraService : public BnCameraService
{
class Client : public BnCamera {};
wp<Client>                    mClient;
}
```

在 CameraService 中，静态函数 instantiate()用于初始化一个 Camera 服务，此函数的代码如下。

```
void CameraService::instantiate() {
defaultServiceManager()->addService( String16("media.camera"), new CameraService());
}
```

其实函数 CameraService::instantiate()注册了一个名称为 media.camera 的服务，此服务和文件 Camera.cpp 中调用的名称相对应。

Camera 整个运作机制是：在文件 Camera.cpp 中调用 ICameraService 的接口，此时实际上调用的是 BpCameraService。而 BpCameraService 通过 Binder 机制和 BnCameraService 实现两个进程的通信。因为 BpCameraService 的实现就是此处的 CameraService，所以 Camera.cpp 虽然是在另外一个进程中运行，但是调用 ICameraService 的接口就像直接调用一样，从函数 connect()中可以得到一个 ICamera 类型的指针，整个指针的实现实际上是 CameraService::Client。

上述 Camera 功能的具体实现就是 CameraService::Client 所实现的，其构造函数如下。

```
CameraService::Client::Client(const sp<CameraService>& cameraService,
        const sp<ICameraClient>& cameraClient) :
mCameraService(cameraService), mCameraClient(cameraClient), mHardware(0)
{
mHardware = openCameraHardware();
mHasFrameCallback = false;
}
```

在构造函数中，通过调用 openCameraHardware()得到一个 CameraHardwareInterface 类型的指针，并作为其成员 mHardware。以后对实际的 Camera 的操作都通过这个指针进行，这是一个简单的直接调用关系。

其实真正的 Camera 功能已经通过实现 CameraHardwareInterface 类来完成。在这个库中，文件

CameraHardwareStub.h 和 CameraHardwareStub.cpp 定义了一个“桩”模块的接口，可以在没有 Camera 硬件的情况下使用。例如在仿真器的情况下使用的文件就是文件 CameraHardwareStub.cpp 和它依赖的文件 FakeCamera.cpp。

类 CameraHardwareStub 的结构如下所示。

```
class CameraHardwareStub : public CameraHardwareInterface {
class PreviewThread : public Thread {
};
};
```

在类 CameraHardwareStub 中包含了线程类 PreviewThread，此线程可以处理 PreView，即负责刷新取景器的内容。实际的 Camera 硬件接口通常可以通过对 V4L2 捕获驱动的调用来实现，同时还需要一个 JPEG 编码程序将从驱动中取出的数据编码成 JPEG 文件。

在文件 FakeCamera.h 和 FakeCamera.cpp 中实现了类 FakeCamera，用于实现一个假的摄像头输入数据的内存，定义代码如下。

```
class FakeCamera {
public:
    FakeCamera(int width, int height);
    ~FakeCamera();

    void setSize(int width, int height);
    void getNextFrameAsRgb565(uint16_t *buffer);//获取 RGB565 格式的预览帧
    void getNextFrameAsYuv422(uint8_t *buffer);//获取 Yuv422 格式的预览帧
    status_t dump(int fd, const Vector<String16>& args);

private:
    void drawSquare(uint16_t *buffer, int x, int y, int size, int color, int shadow);
    void drawCheckerboard(uint16_t *buffer, int size);

    static const int kRed = 0xf800;
    static const int kGreen = 0x07c0;
    static const int kBlue = 0x003e;

    int           mWidth, mHeight;
    int           mCounter;
    int           mCheckX, mCheckY;
    uint16_t      *mTmpRgb16Buffer;
};
```

当在 CameraHardwareStub 中设置参数后会调用函数 initHeapLocked()，此函数的实现代码如下。

```
void CameraHardwareStub::initHeapLocked()
{
    int picture_width, picture_height;
    mParameters.getPictureSize(&picture_width, &picture_height);
    //建立内存堆栈，创建两块内存
    mRawHeap = new MemoryHeapBase(picture_width * 2 * picture_height);

    int preview_width, preview_height;
    mParameters.getPreviewSize(&preview_width, &preview_height);
    LOGD("initHeapLocked: preview size=%dx%d", preview_width, preview_height);

    //从参数中获取信息
```

```
        int how_big = preview_width * preview_height * 2;

        // If we are being reinitialized to the same size as before, no
        // work needs to be done.
        if (how_big == mPreviewFrameSize)
            return;

        mPreviewFrameSize = how_big;

        // Make a new mmap'ed heap that can be shared across processes.
        // use code below to test with pmem
        mPreviewHeap = new MemoryHeapBase(mPreviewFrameSize * kBufferCount);
        //建立内存队列
        for (int i = 0; i < kBufferCount; i++) {
            mBuffers[i] = new MemoryBase(mPreviewHeap, i * mPreviewFrameSize, mPreviewFrameSize);
        }

        // Recreate the fake camera to reflect the current size.
        delete mFakeCamera;
        mFakeCamera = new FakeCamera(preview_width, preview_height);
}
```

定义函数 startPreview()创建一个线程，此函数的实现代码如下。

```
status_t CameraHardwareStub::startPreview(preview_callback cb, void* user)
{
        Mutex::Autolock lock(mLock);
        if (mPreviewThread != 0) {
            // already running
            return INVALID_OPERATION;
        }
        mPreviewCallback = cb;
        mPreviewCallbackCookie = user;
        mPreviewThread = new PreviewThread(this);//建立视频预览线程
        return NO_ERROR;
}
```

通过上面建立的线程可以调用预览回调机制，将预览的数据传递给上层的 CameraService。

创建预览线程函数 previewThread()，建立一个循环以得到假的摄像头输入数据的来源，并通过预览回调函数将输出传到上层中。函数 previewThread()的主要实现代码如下。

```
int CameraHardwareStub::previewThread()
{
        mLock.lock();
           int previewFrameRate = mParameters.getPreviewFrameRate();
            //发现在当前缓冲的堆之内的垂距
            ssize_t offset = mCurrentPreviewFrame * mPreviewFrameSize;
            sp<MemoryHeapBase> heap = mPreviewHeap;
            //假设假照相机内部状态没有变化
            // (or is thread safe)
            FakeCamera* fakeCamera = mFakeCamera;
            sp<MemoryBase> buffer = mBuffers[mCurrentPreviewFrame];
                mLock.unlock();
        if (buffer != 0) {
            //计算在框架之间等待多久时间
```

```
            int delay = (int)(1000000.0f / float(previewFrameRate));
                //这总是合法的，即使内存消亡仍然在我们的过程中被映射
                void *base = heap->base();
                //用假照相机填装当前框架
            uint8_t *frame = ((uint8_t *)base) + offset;
            fakeCamera->getNextFrameAsYuv422(frame);

                        // Notify the client of a new frame.
            mPreviewCallback(buffer, mPreviewCallbackCookie);
                //推送进缓冲区实现预览
            mCurrentPreviewFrame = (mCurrentPreviewFrame + 1) % kBufferCount;
            //等待它
            usleep(delay);
        }
        return NO_ERROR;
}
```

在上述文件中还定义了其他的函数，函数的功能一看便知，在此为节省篇幅不再一一进行详细讲解，请读者参考开源的代码文件。

21.2　移植 Camera 系统

在 Linux 系统中，Camera 驱动程序使用了 Linux 标准的 Video for Linux 2（V4L2）驱动程序。无论是内核空间还是用户空间，都使用 V4L2 驱动程序框架来定义数据类和控制类。所以在移植 Android 中的 Camera 系统时，也是用标准的 V4L2 驱动程序作为 Camera 的驱动程序。

在 Android 系统中，Camera 系统的标准化部分是硬件抽象层接口，所以我们在某平台移植 Camera 系统时，主要工作是移植 Camera 驱动程序和 Camera 硬件抽象层。Camera 的硬件抽象层是 V4L2 和 CameraService 之间的接口，是一个 C++接口类，我们需要具体的实现者来继承这个类，并且实现里面的虚函数。Camera 的硬件抽象层需要具备取景器、视频录制、相片拍摄等功能。在 Camera 系统中，具体任务分配如下所示。

☑　V4L2 驱动程序：任务是获得 Video 数据。
☑　Camera 的硬件抽象层：任务是将纯视频流和取景器、实现预览、向上层发送数据等功能组织起来。
☑　其他算法库和硬件：任务是实现自动对焦和成像增强等功能。

21.2.1　实现 V4L2 驱动

在 Linux 系统中，Camera 驱动程序使用了 Linux 标准的 Video for Linux 2（V4L2）驱动程序。无论是内核空间还是用户空间，都使用 V4L2 驱动程序框架来定义数据类和控制类。所以在移植 Android 中的 Camera 系统时，也是用标准的 V4L2 驱动程序作为 Camera 的驱动程序。在 Camera 系统中，V4L2 驱动程序的任务是获得 Video 数据。

1．V4L2 API

V4L2 是 V4L 的升级版本，为 Linux 下视频设备程序提供了一套接口规范，包括一套数据结构和底层 V4L2 驱动接口。V4L2 驱动程序向用户空间提供字符设备，主设备号是 81，对于视频设备来说，次设备号是 0～63。如果次设备号在 64～127 之间则是 Radio 设备，次设备号在 192～223 之间则是 Teletext 设备，次

设备号在 224～255 之间则是 VBI 设备。V4L2 中常用的结构体在内核文件 include/linux/videodev2.h 中定义，代码如下所示。

```
struct v4l2_requestbuffers                    //申请帧缓冲，对应命令 VIDIOC_REQBUFS
        struct v4l2_capability                //视频设备的功能，对应命令 VIDIOC_QUERYCAP
        struct v4l2_input                     //视频输入信息，对应命令 VIDIOC_ENUMINPUT
        struct v4l2_standard                  //视频的制式，例如 PAL、NTSC，对应命令 VIDIOC_ENUMSTD
        struct v4l2_format                    //帧的格式，对应命令 VIDIOC_G_FMT、VIDIOC_S_FMT 等
        struct v4l2_buffer                    //驱动中的一帧图像缓存，对应命令 VIDIOC_QUERYBUF
        struct v4l2_crop                      //视频信号矩形边框
        v4l2_std_id                           //视频制式
```

常用的 ioctl 接口命令也在文件 include/linux/videodev2.h 中定义，代码如下所示。

```
VIDIOC_REQBUFS                                //分配内存
        VIDIOC_QUERYBUF                       //把 VIDIOC_REQBUFS 中分配的数据缓存转换成物理地址
        VIDIOC_QUERYCAP                       //查询驱动功能
        VIDIOC_ENUM_FMT                       //获取当前驱动支持的视频格式
        VIDIOC_S_FMT                          //设置当前驱动的视频捕获格式
        VIDIOC_G_FMT                          //读取当前驱动的视频捕获格式
        VIDIOC_TRY_FMT                        //验证当前驱动的显示格式
        VIDIOC_CROPCAP                        //查询驱动的修剪能力
        VIDIOC_S_CROP                         //设置视频信号的矩形边框
        VIDIOC_G_CROP                         //读取视频信号的矩形边框
        VIDIOC_QBUF                           //把数据从缓存中读取出来
        VIDIOC_DQBUF                          //把数据放回缓存队列
        VIDIOC_STREAMON                       //开始视频显示函数
        VIDIOC_STREAMOFF                      //结束视频显示函数
        VIDIOC_QUERYSTD                       //检查当前视频设备支持的标准，例如 PAL 或 NTSC
```

2．操作 V4L2 的流程

在 V4L2 中提供了很多访问接口，我们可以根据具体需要选择操作方法。需要注意的是，很少有驱动完全实现了所有的接口功能。所以在使用时需要参考驱动源码，或仔细阅读驱动提供者的使用说明。接下来简单列举出一种 V4L2 的操作流程供读者参考。

（1）打开设备文件，具体代码如下。

```
int fd = open(Devicename,mode);
Devicename：/dev/video0、/dev/video1 …
Mode：O_RDWR [| O_NONBLOCK]
```

如果需要使用非阻塞模式调用视频设备，当没有可用的视频数据时不会阻塞而会立刻返回。

（2）获取设备的 capability，具体代码如下。

```
struct v4l2_capability capability;
int ret = ioctl(fd, VIDIOC_QUERYCAP, &capability);
```

在此需要查看设备具有什么功能，例如是否具有视频输入特性。

（3）选择视频输入，代码如下。

```
struct v4l2_input input;
//开始初始化 Input
int ret = ioctl(fd, VIDIOC_QUERYCAP, &input);
```

每一个视频设备可以有多个视频输入，如果只有一路输入，则可以没有这个功能。

（4）检测视频支持的制式，具体代码如下。

```
v4l2_std_id std;
do {
```

```
                ret = ioctl(fd, VIDIOC_QUERYSTD, &std);
 } while (ret == -1 && errno == EAGAIN);
                  switch (std) {
                  case V4L2_STD_NTSC:
                ...
                  case V4L2_STD_PAL:
                ...
                  }
```

（5）设置视频捕获格式，具体代码如下。

```
struct v4l2_format fmt;
fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.height = height;
fmt.fmt.pix.width = width;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if(ret) {
        perror("VIDIOC_S_FMT\n");
        close(fd);
        return -1;
}
```

（6）向驱动申请帧缓存，具体代码如下。

```
struct v4l2_requestbuffers req;
 if (ioctl(fd, VIDIOC_REQBUFS, &req) == -1) {
        return -1;
 }
```

在结构 v4l2_requestbuffers 中定义了缓存的数量，驱动会根据这个申请对应数量的视频缓存。通过多个缓存可以建立 FIFO，这样可以提高视频采集的效率。

（7）获取每个缓存的信息，并 mmap 到用户空间，主要代码如下。

```
typedef struct VideoBuffer {
        void *start;
        size_t length;
} VideoBuffer;
          VideoBuffer* buffers = calloc( req.count, sizeof(*buffers) );
struct v4l2_buffer buf;
          for (numBufs = 0; numBufs < req.count; numBufs++) {//映射所有的缓存
        memset( &buf, 0, sizeof(buf) );
        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_MMAP;
        buf.index = numBufs;
        if (ioctl(fd, VIDIOC_QUERYBUF, &buf) == -1) {//获取到对应 index 的缓存信息，此处主要利用 length 信
息及 offset 信息来完成后面的 mmap 操作
                return -1;
        }
buffers[numBufs].length = buf.length;
        //转换成相对地址
        buffers[numBufs].start = mmap(NULL, buf.length,
                PROT_READ | PROT_WRITE,
                MAP_SHARED,
                fd, buf.m.offset);
```

```
if (buffers[numBufs].start == MAP_FAILED) {
                    return -1;
          }
```

（8）开始采集视频，具体代码如下。

```
int buf_type= V4L2_BUF_TYPE_VIDEO_CAPTURE;
int ret = ioctl(fd, VIDIOC_STREAMON, &buf_type);
```

（9）取出 FIFO 缓存中已经采样的帧缓存，具体代码如下。

```
struct v4l2_buffer buf;
memset(&buf,0,sizeof(buf));
buf.type=V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory=V4L2_MEMORY_MMAP;
buf.index=0;//此值由下面的 ioctl 返回
if (ioctl(fd, VIDIOC_DQBUF, &buf) == -1)
{
          return -1;
}
```

通过上述代码，可以根据返回的 buf.index 找到对应的 mmap 映射好的缓存，实现取出视频数据的功能。

（10）将刚刚处理完的缓冲重新入队列尾，这样可以循环采集，具体代码如下。

```
if (ioctl(fd, VIDIOC_QBUF, &buf) == -1) {
          return -1;
}
```

（11）停止视频的采集，具体代码如下。

```
int ret = ioctl(fd, VIDIOC_STREAMOFF, &buf_type);
```

（12）关闭视频设备，具体代码如下。

```
close(fd);
```

3. V4L2 驱动框架

在上述使用 V4L2 的流程中，各个操作都需要有底层 V4L2 驱动的支持。在内核中有一些非常完善的例子。例如在 Linux-2.6.26 内核目录/drivers/media/video//zc301/中，文件 zc301_core.c 实现了 ZC301 视频驱动代码。

（1）V4L2 驱动注册、注销函数

在 Video 核心层文件 drivers/media/video/videodev.c 中提供了注册函数，具体代码如下。

```
int video_register_device(struct video_device *vfd, int type, int nr)
```

☑ video_device：要构建的核心数据结构。

☑ type：表示设备类型，此设备号的基地址受此变量的影响。

☑ nr：如果 end-base>nr>0，次设备号=base（基准值，受 type 影响）+nr；否则系统将自动分配合适的次设备号。

我们具体需要的驱动只需构建 video_device 结构，然后调用注册函数即可。例如在文件 zc301_core.c 中的实现代码如下。

```
err = video_register_device(cam->v4ldev, VFL_TYPE_GRABBER,
                                        video_nr[dev_nr]);
```

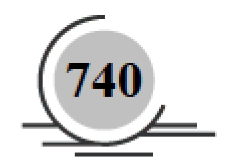
在 Video 核心层文件 drivers/media/video/videodev.c 中提供了如下注销函数。

```
void video_unregister_device(struct video_device *vfd)
```

（2）构建 struct video_device

在结构 video_device 中包含了视频设备的属性和操作方法，具体可以参考文件 zc301_core.c，代码如下。

```
strcpy(cam->v4ldev->name, "ZC0301[P] PC Camera");
  cam->v4ldev->owner = THIS_MODULE;
```



```
cam->v4ldev->type = VID_TYPE_CAPTURE | VID_TYPE_SCALES;
cam->v4ldev->fops = &zc0301_fops;
cam->v4ldev->minor = video_nr[dev_nr];
cam->v4ldev->release = video_device_release;
video_set_drvdata(cam->v4ldev, cam);
```

在上述 zc301 的驱动中并没有实现 struct video_device 中的很多操作函数，例如 vidioc_querycap、vidioc_g_fmt_cap，这是因为在 struct file_operations zc0301_fops 中的 zc0301_ioctl 实现了前面的所有 ioctl 操作，所以无须在 struct video_device 再次实现 struct video_device 中的操作。

另外也可以使用下面的代码来构建 struct video_device。

```
static struct video_device camif_dev =
  {
    .name = "s3c2440 camif",
    .type = VID_TYPE_CAPTURE|VID_TYPE_SCALES|VID_TYPE_SUBCAPTURE,
    .fops = &camif_fops,
    .minor = -1,
    .release = camif_dev_release,
    .vidioc_querycap = vidioc_querycap,
    .vidioc_enum_fmt_cap = vidioc_enum_fmt_cap,
    .vidioc_g_fmt_cap = vidioc_g_fmt_cap,
    .vidioc_s_fmt_cap = vidioc_s_fmt_cap,
    .vidioc_queryctrl = vidioc_queryctrl,
    .vidioc_g_ctrl = vidioc_g_ctrl,
    .vidioc_s_ctrl = vidioc_s_ctrl,
  };
  static struct file_operations camif_fops =
  {
    .owner = THIS_MODULE,
    .open = camif_open,
    .release = camif_release,
    .read = camif_read,
    .poll = camif_poll,
    .ioctl = video_ioctl2, /* V4L2 ioctl handler */
    .mmap = camif_mmap,
    .llseek = no_llseek,
  };
```

结构 video_ioctl2 是在文件 videodev.c 中实现的，video_ioctl2 会根据 ioctl 不同的 cmd 来调用 video_device 中的操作方法。

4．实现 Video 核心层

具体实现代码可参考内核文件/drivers/media/videodev.c，实现流程如下。

（1）注册 256 个视频设备，代码如下。

```
static int __init videodev_init(void)
        {
                int ret;
                if (register_chrdev(VIDEO_MAJOR, VIDEO_NAME, &video_fops)) {
                        return -EIO;
                }
                ret = class_register(&video_class);
                ...
        }
```

在上述代码中注册了 256 个视频设备和 video_class 类，video_fops 类为这 256 个设备共同的操作方法。

（2）实现 V4L2 驱动的注册函数，具体代码如下。

```
int video_register_device(struct video_device *vfd, int type, int nr)
        {
                int i=0;
                int base;
                int end;
                int ret;
                char *name_base;
        switch(type) //根据不同的 type 确定设备名称、次设备号
                {
                        case VFL_TYPE_GRABBER:
                                base=MINOR_VFL_TYPE_GRABBER_MIN;
                                end=MINOR_VFL_TYPE_GRABBER_MAX+1;
                                name_base = "video";
                                break;
                        case VFL_TYPE_VTX:
                                base=MINOR_VFL_TYPE_VTX_MIN;
                                end=MINOR_VFL_TYPE_VTX_MAX+1;
                                name_base = "vtx";
                                break;
                        case VFL_TYPE_VBI:
                                base=MINOR_VFL_TYPE_VBI_MIN;
                                end=MINOR_VFL_TYPE_VBI_MAX+1;
                                name_base = "vbi";
                                break;
                        case VFL_TYPE_RADIO:
                                base=MINOR_VFL_TYPE_RADIO_MIN;
                                end=MINOR_VFL_TYPE_RADIO_MAX+1;
                                name_base = "radio";
                                break;
                        default:
                                printk(KERN_ERR "%s called with unknown type: %d\n",
                                        __func__, type);
                                return -1;
                }
        /*  计算出次设备号  */
                mutex_lock(&videodev_lock);
                if (nr >= 0 && nr < end-base) {
                        /* use the one the driver asked for */
                        i = base+nr;
                        if (NULL != video_device[i]) {
                                mutex_unlock(&videodev_lock);
                                return -ENFILE;
                        }
                } else {
                        /* use first free */
                        for(i=base;i<end;i++)
                                if (NULL == video_device[i])
                                        break;
                        if (i == end) {
```

```
                    mutex_unlock(&videodev_lock);
                    return -ENFILE;
                }
            }
        video_device[i]=vfd; //保存 video_device 结构指针到系统的结构数组中，最终的次设备号和 i 相关
        vfd->minor=i;
        mutex_unlock(&videodev_lock);
        mutex_init(&vfd->lock);
    /* sysfs class */
        memset(&vfd->class_dev, 0x00, sizeof(vfd->class_dev));
        if (vfd->dev)
            vfd->class_dev.parent = vfd->dev;
        vfd->class_dev.class = &video_class;
        vfd->class_dev.devt = MKDEV(VIDEO_MAJOR, vfd->minor);
        sprintf(vfd->class_dev.bus_id, "%s%d", name_base, i - base);//最后在/dev 目录下的名称
    ret = device_register(&vfd->class_dev);//结合 udev 或 mdev 可以实现自动在/dev 下创建设备节点
        ...
    }
```

从上面的注册函数代码中可以看出，注册 V4L2 驱动的过程只是创建了设备节点，例如/dev/video0，并且保存了 video_device 结构指针。

（3）打开视频驱动。

当使用下面的代码在用户空间调用 open()函数打开对应的视频文件时：

```
int fd = open(/dev/video0, O_RDWR);
```

对应/dev/video0 目录的文件操作结构是在文件/drivers/media/videodev.c 中定义的 video_fops，代码如下。

```
static const struct file_operations video_fops=
    {
        .owner = THIS_MODULE,
        .llseek = no_llseek,
        .open = video_open,
    };
```

上述代码只是实现了 open 操作，后面的其他操作需要使用 video_open()来实现，具体代码如下。

```
static int video_open(struct inode *inode, struct file *file)
    {
        unsigned int minor = iminor(inode);
        int err = 0;
        struct video_device *vfl;
        const struct file_operations *old_fops;
    if(minor>=VIDEO_NUM_DEVICES)
            return -ENODEV;
        mutex_lock(&videodev_lock);
        vfl=video_device[minor];
        if(vfl==NULL) {
            mutex_unlock(&videodev_lock);
            request_module("char-major-%d-%d", VIDEO_MAJOR, minor);
            mutex_lock(&videodev_lock);
            vfl=video_device[minor]; //根据次设备号取出 video_device 结构
            if (vfl==NULL) {
                mutex_unlock(&videodev_lock);
                return -ENODEV;
            }
```

```
            }
            old_fops = file->f_op;
            //替换此打开文件的 file_operation 结构。后面其他针对此文件的操作都由新的结构来负责，也
就是由每个具体的 video_device 的 fops 负责
            file->f_op = fops_get(vfl->fops);
            if(file->f_op->open)
                err = file->f_op->open(inode,file);
            if (err) {
                fops_put(file->f_op);
                file->f_op = fops_get(old_fops);
            }
            ...
    }
```

21.2.2 实现硬件抽象层

在 Andorid 2.1 及其以前的版本中，Camera 系统的硬件抽象层的头文件保存在目录 frameworks/base/include/ui/中。

在 Andorid 2.2 及其以后的版本中，Camera 系统的硬件抽象层的头文件保存在目录 frameworks/av/include/camera/中。

在上述目录中主要包含了如下头文件。

☑ CameraHardwareInterface.h：在里面定义了 C++接口类，此类需要根据系统的情况实现继承。

☑ CameraParameters.h：在里面定义了 Camera 系统的参数，可以在本地系统的各个层次中使用这些参数。

☑ Camera.h：在里面提供了 Camera 系统本地对上层的接口。

1．Andorid 2.1 及其以前的版本

在 Andorid 2.1 及其以前的版本中，在文件 CameraHardwareInterface.h 中首先定义了硬件抽象层接口的回调函数类型，对应代码如下。

```
/** startPreview()使用的回调函数*/
typedef void (*preview_callback)(const sp<IMemory>& mem, void* user);

/** startRecord()使用的回调函数*/
typedef void (*recording_callback)(const sp<IMemory>& mem, void* user);

/** takePicture()使用的回调函数*/
typedef void (*shutter_callback)(void* user);

/** takePicture()使用的回调函数*/
typedef void (*raw_callback)(const sp<IMemory>& mem, void* user);

/** takePicture()使用的回调函数*/
typedef void (*jpeg_callback)(const sp<IMemory>& mem, void* user);

/** autoFocus()使用的回调函数*/
typedef void (*autofocus_callback)(bool focused, void* user);
```

然后定义类 CameraHardwareInterface，并在类中定义了各个接口函数，具体代码如下。

```
class CameraHardwareInterface : public virtual RefBase {
public:
```

```
    virtual ~CameraHardwareInterface() { }
    virtual sp<IMemoryHeap>           getPreviewHeap() const = 0;
    virtual sp<IMemoryHeap>           getRawHeap() const = 0;
    virtual status_t      startPreview(preview_callback cb, void* user) = 0;
    virtual bool useOverlay() {return false;}
    virtual status_t setOverlay(const sp<Overlay> &overlay) {return BAD_VALUE;}
    virtual void          stopPreview() = 0;
    virtual bool          previewEnabled() = 0;
    virtual status_t      startRecording(recording_callback cb, void* user) = 0;
    virtual void          stopRecording() = 0;
    virtual bool          recordingEnabled() = 0;
    virtual void          releaseRecordingFrame(const sp<IMemory>& mem) = 0;
    virtual status_t      autoFocus(autofocus_callback,
                                    void* user) = 0;
    virtual status_t      takePicture(shutter_callback,
                                      raw_callback,
                                      jpeg_callback,
                                      void* user) = 0;
    virtual status_t      cancelPicture(bool cancel_shutter,
                                        bool cancel_raw,
                                        bool cancel_jpeg) = 0;
    /** Return the camera parameters. */
    virtual CameraParameters   getParameters() const = 0;
    virtual void release() = 0;
    virtual status_t dump(int fd, const Vector<String16>& args) const = 0;
};
extern "C" sp<CameraHardwareInterface> openCameraHardware();
};
```

可以将上述代码中的接口分为如下几类。

☑ 取景预览：startPreview、stopPreview、useOverlay 和 setOverlay。

☑ 录制视频：startRecording、stopRecording、recordingEnabled 和 releaseRecordingFrame。

☑ 拍摄照片：takePicture 和 cancelPicture。

☑ 辅助功能：autoFocus（自动对焦）、setParameters 和 getParameters。

2．Andorid 2.2 及其以后的版本

在 Andorid 2.2 及其以后的版本中，在文件 Camera.h 中首先定义了通知信息的枚举值，对应代码如下所示。

```
enum {
    CAMERA_MSG_ERROR                = 0x001,      //错误信息
    CAMERA_MSG_SHUTTER              = 0x002,      //快门信息
    CAMERA_MSG_FOCUS                = 0x004,      //聚焦信息
    CAMERA_MSG_ZOOM                 = 0x008,      //缩放信息
    CAMERA_MSG_PREVIEW_FRAME        = 0x010,      //帧预览信息
    CAMERA_MSG_VIDEO_FRAME          = 0x020,      //视频帧信息
    CAMERA_MSG_POSTVIEW_FRAME       = 0x040,      //拍照后停止帧信息
    CAMERA_MSG_RAW_IMAGE            = 0x080,      //原始数据格式照片信息
    CAMERA_MSG_COMPRESSED_IMAGE = 0x100,      //压缩格式照片信息
    CAMERA_MSG_ALL_MSGS             = 0x1FF       //所有信息
};
```

然后在文件 CameraHardwareInterface.h 中定义如下 3 个回调函数。

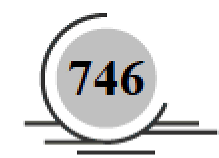
```
//通知回调
typedef void (*notify_callback)(int32_t msgType,
                                int32_t ext1,
                                int32_t ext2,
                                void* user);
//数据回调
typedef void (*data_callback)(int32_t msgType,
                              const sp<IMemory>& dataPtr,
                              void* user);
//带有时间戳的数据回调
typedef void (*data_callback_timestamp)(nsecs_t timestamp,
                                        int32_t msgType,
                                        const sp<IMemory>& dataPtr,
                                        void* user);
```

然后定义类 CameraHardwareInterface，在类中的各个函数的具体实现和其他 Android 版本中相同，区别是回调函数不再由各个函数分别设置，所以在函数 startPreview()和 startRecording()中缺少了回调函数的指针和 void*类型的附加参数，主要实现代码如下。

```
class CameraHardwareInterface : public virtual RefBase {
public:
    virtual ~CameraHardwareInterface() { }
    virtual sp<IMemoryHeap>         getPreviewHeap() const = 0;
    virtual sp<IMemoryHeap>         getRawHeap() const = 0;
    virtual void setCallbacks(notify_callback notify_cb,
                              data_callback data_cb,
                              data_callback_timestamp data_cb_timestamp,
    virtual void          enableMsgType(int32_t msgType) = 0;
    virtual void          disableMsgType(int32_t msgType) = 0;
    virtual bool          msgTypeEnabled(int32_t msgType) = 0;
    virtual status_t      startPreview() = 0;
    virtual status_t      getBufferInfo(sp<IMemory>& Frame, size_t *alignedSize) = 0;
    virtual bool          useOverlay() {return false;}
    virtual status_t      setOverlay(const sp<Overlay> &overlay) {return BAD_VALUE;}
    virtual void          stopPreview() = 0;
    virtual bool          previewEnabled() = 0;
    virtual status_t      startRecording() = 0;
    virtual void          stopRecording() = 0;
    virtual bool          recordingEnabled() = 0;
    virtual void          releaseRecordingFrame(const sp<IMemory>& mem) = 0;
    virtual status_t      autoFocus() = 0;
    virtual status_t      cancelAutoFocus() = 0;
    virtual status_t      takePicture() = 0;
    virtual status_t      cancelPicture() = 0;
    virtual CameraParameters   getParameters() const = 0;
    virtual status_t sendCommand(int32_t cmd, int32_t arg1, int32_t arg2) = 0;
    virtual void release() = 0;
    virtual status_t d
}
```

因为在新版本的 Camera 系统中增加了 sendCommand()，所以需要在文件 Camera.h 中增加新命令和返回值，具体实现代码如下。

```
//函数 sendCommand()使用的命令类型
enum {
```



```
    CAMERA_CMD_START_SMOOTH_ZOOM = 1,
    CAMERA_CMD_STOP_SMOOTH_ZOOM = 2,
    CAMERA_CMD_SET_DISPLAY_ORIENTATION = 3,
};

//错误类型
enum {
    CAMERA_ERROR_UKNOWN = 1,
    CAMERA_ERROR_SERVER_DIED = 100
};
```

3．实现 Camera 硬件抽象层

在函数 startPreview()的实现过程中，保存预览回调函数并建立了预览线程机制。在预览线程的循环中等待视频数据的到达，当视频帧到达后调用预览回调函数传出视频帧。

在 Camera 硬件抽象层中，实现取景器预览功能的主要步骤如下。

（1）在初始化的过程中，建立预览数据的内存队列（多种方式）。

（2）在函数 startPreview()中建立预览线程。

（3）在预览线程的循环中，等待视频数据到达。

（4）视频到达后使用预览回调机制将视频向上传送。

在此过程不需要使用预览回调函数，可以直接将视频数据输入到 Overlay 上。如果使用 Overlay 实现取景器，则需要有以下两个变化。

☑　在函数 setOverlay()中，从 ISurface 接口中取得 Overlay 类。

☑　在预览线程的循环中，不是用预览回调函数直接将数据输入到 Overlay 上。

在 Camera 硬件抽象层中，录制视频的主要步骤如下。

（1）在函数 startRecording()的实现（或者在 setCallbacks）中保存录制视频回调函数。

（2）录制视频可以使用自己的线程，也可以使用预览线程。

（3）通过调用录制回调函数的方式将视频帧送出。

当调用函数 releaseRecordingFrame()后，表示上层通知 Camera 硬件抽象层，这一帧的内存已经用完，可以进行下一次的处理。如果在 V4L2 驱动程序中使用原始数据（RAW），则视频录制的数据和取景器预览的数据为同一数据。当调用 releaseRecordingFrame()时，通常表示编码器已经完成了对当前视频帧的编码，对这块内存进行释放。在这个函数的实现中，可以设置标志位，标记帧内存可以再次使用。

由此可见，对于 Linux 系统来说，摄像头驱动部分大多使用 Video for Linux 2（V4L2）驱动程序，在此处主要的处理流程如下。

（1）如果使用映射内核内存的方式（V4L2_MEMORY_MMAP），则构建预览的内存 MemoryHeapBase 需要从 V4L2 驱动程序中得到内存指针。

（2）如果使用用户空间内存的方式（V4L2_MEMORY_USERPTR），则 MemoryHeapBase 中开辟的内存是在用户空间建立的。

（3）在预览的线程中，使用 VIDIOC_DQBUF 调用阻塞等待视频帧的到来，处理完成后使用 VIDIOC_QBUF 调用将帧内存再次压入队列，然后等待下一帧的到来。

21.3　实战演练——在 MSM 平台实现 Camera 驱动

在 MSM 平台中，和 Camera 系统相关的文件如下。

☑ drivers/media/video/msm/msm_v4l2.c：是 V4L2 驱动程序的入口文件。

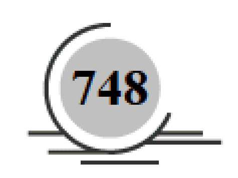
☑ drivers/media/video/msm/msm_camera.c：是公用库函数。

☑ drivers/media/video/msm/s5k3e2fx.c：摄像头传感器驱动文件，使用 I2C 接口控制。

文件 msm_camera.h 是和摄像头相关的头文件，在里面定义了各种额外的 ioctl 命令，其主要代码如下。

```
#define MSM_CAM_IOCTL_MAGIC 'm'
#define MSM_CAM_IOCTL_GET_SENSOR_INFO    _IOR(MSM_CAM_IOCTL_MAGIC, 1, struct msm_camsensor_
info *)
#define MSM_CAM_IOCTL_REGISTER_PMEM    _IOW(MSM_CAM_IOCTL_MAGIC, 2, struct msm_pmem_
info *)
#define MSM_CAM_IOCTL_UNREGISTER_PMEM    _IOW(MSM_CAM_IOCTL_MAGIC, 3, unsigned)
#define MSM_CAM_IOCTL_CTRL_COMMAND    _IOW(MSM_CAM_IOCTL_MAGIC, 4, struct msm_ctrl_cmd *)
#define MSM_CAM_IOCTL_CONFIG_VFE    _IOW(MSM_CAM_IOCTL_MAGIC, 5, struct msm_camera_vfe_
cfg_cmd *)
#define MSM_CAM_IOCTL_GET_STATS    _IOR(MSM_CAM_IOCTL_MAGIC, 6, struct msm_camera_stats_
event_ctrl *)
#define MSM_CAM_IOCTL_GETFRAME    _IOR(MSM_CAM_IOCTL_MAGIC, 7, struct msm_camera_get_
frame *)
#define MSM_CAM_IOCTL_ENABLE_VFE    _IOW(MSM_CAM_IOCTL_MAGIC, 8, struct camera_enable_
cmd *)
#define MSM_CAM_IOCTL_CTRL_CMD_DONE    _IOW(MSM_CAM_IOCTL_MAGIC, 9, struct camera_cmd *)
#define MSM_CAM_IOCTL_CONFIG_CMD    _IOW(MSM_CAM_IOCTL_MAGIC, 10, struct camera_cmd *)
#define MSM_CAM_IOCTL_DISABLE_VFE    _IOW(MSM_CAM_IOCTL_MAGIC, 11, struct camera_enable_
cmd *)
#define MSM_CAM_IOCTL_PAD_REG_RESET2    _IOW(MSM_CAM_IOCTL_MAGIC, 12, struct camera_
enable_cmd *)
#define MSM_CAM_IOCTL_VFE_APPS_RESET    _IOW(MSM_CAM_IOCTL_MAGIC, 13, struct camera_
enable_cmd *)
#define MSM_CAM_IOCTL_RELEASE_FRAME_BUFFER    _IOW(MSM_CAM_IOCTL_MAGIC, 14, struct
camera_enable_cmd *)
#define MSM_CAM_IOCTL_RELEASE_STATS_BUFFER    _IOW(MSM_CAM_IOCTL_MAGIC, 15, struct
msm_stats_buf *)
#define MSM_CAM_IOCTL_AXI_CONFIG    _IOW(MSM_CAM_IOCTL_MAGIC, 16, struct msm_camera_vfe_
cfg_cmd *)
#define MSM_CAM_IOCTL_GET_PICTURE    _IOW(MSM_CAM_IOCTL_MAGIC, 17, struct msm_camera_
ctrl_cmd *)
#define MSM_CAM_IOCTL_SET_CROP    _IOW(MSM_CAM_IOCTL_MAGIC, 18, struct crop_info *)
#define MSM_CAM_IOCTL_PICT_PP    _IOW(MSM_CAM_IOCTL_MAGIC, 19, uint8_t *)
#define MSM_CAM_IOCTL_PICT_PP_DONE    _IOW(MSM_CAM_IOCTL_MAGIC, 20, struct msm_snapshot_
pp_status *)
#define MSM_CAM_IOCTL_SENSOR_IO_CFG    _IOW(MSM_CAM_IOCTL_MAGIC, 21, struct sensor_cfg_
data *)
#define MSM_CAMERA_LED_OFF 0
#define MSM_CAMERA_LED_LOW 1
#define MSM_CAMERA_LED_HIGH 2
#define MSM_CAM_IOCTL_FLASH_LED_CFG    _IOW(MSM_CAM_IOCTL_MAGIC, 22, unsigned *)
#define MSM_CAM_IOCTL_UNBLOCK_POLL_FRAME    _IO(MSM_CAM_IOCTL_MAGIC, 23)
#define MSM_CAM_IOCTL_CTRL_COMMAND_2    _IOW(MSM_CAM_IOCTL_MAGIC, 24, struct msm_ctrl_
cmd *)
```

文件 msm_camera.c 辅助实现 Camera 系统的功能，里面包含了供内核调用的文件，也提供了给用户空间的接口。其中在用户空间的设备节点就是 dev/msm_camera/中的 3 个设备：配置设备 config0、控制设备



control0 和帧数据设备 frame0。上面的 ioctl 命令都是为这些设备节点使用的。

在文件 msm_camera.c 中为内核空间提供接口，主要实现代码如下。

```
int msm_v4l2_register(struct msm_v4l2_driver *drv)//注册 msm_v4l2_driver 驱动
{
		if (list_empty(&msm_sensors))
				return -ENODEV;
		drv->sync = list_first_entry(&msm_sensors, struct msm_sync, list);
		drv->open = __msm_open;
		drv->release = __msm_release;
		drv->ctrl = __msm_v4l2_control;
		drv->reg_pmem = __msm_register_pmem;
		drv->get_frame = __msm_get_frame;
		drv->put_frame = __msm_put_frame_buf;
		drv->get_pict = __msm_get_pic;
		drv->drv_poll = __msm_poll_frame;
		return 0;
}
EXPORT_SYMBOL(msm_v4l2_register);                    //注销 msm_v4l2_driver 驱动
int msm_v4l2_unregister(struct msm_v4l2_driver *drv)
{
		drv->sync = NULL;
		return 0;
}
static int msm_device_init(struct msm_cam_device *pmsm,//开始注册 Camera 驱动
				struct msm_sync *sync,
				int node)
```

MSM 平台中的 Camera 硬件抽象层已经包含在 Android 代码中，此部分的内容保存在如下文件中。

☑　文件 hardware/msm7k/libcamera/camera_ifc.h：定义 Camera 接口中的常量。

☑　文件 hardware/msm7k/libcamera/QualcommCameraHardware.h：是硬件抽象层的头文件。

☑　文件 hardware/msm7k/libcamera/QualcommCameraHardware.cpp：是硬件抽象层的实现。

在文件 QualcommCameraHardware.h 中定义一个表示内存的类 MemPool。在 Android 系统中，类 AshmemPool 和类 PmemPool 都是 MemPool 的继承者，PreviewPmemPool 和 RawPmemPool 是 MemPool 的继承者，具体实现代码如下。

```
struct MemPool : public RefBase {
    MemPool(int buffer_size, int num_buffers,
            int frame_size,
            int frame_offset,
            const char *name);
    virtual ~MemPool() = 0;
    void completeInitialization();
    bool initialized() const {
        return mHeap != NULL && mHeap->base() != MAP_FAILED;
    }
    virtual status_t dump(int fd, const Vector<String16>& args) const;
    int mBufferSize;
    int mNumBuffers;
    int mFrameSize;
    int mFrameOffset;
    sp<MemoryHeapBase> mHeap;
```

```
      sp<MemoryBase> *mBuffers;
      const char *mName;
};
struct AshmemPool : public MemPool {
      AshmemPool(int buffer_size, int num_buffers,
                    int frame_size,
                    int frame_offset,
                    const char *name);
};
struct PmemPool : public MemPool {
      PmemPool(const char *pmem_pool,
                 int buffer_size, int num_buffers,
                 int frame_size,
                 int frame_offset,
                 const char *name);
      virtual ~PmemPool() { }
      int mFd;
      uint32_t mAlignedSize;
      struct pmem_region mSize;
};
struct PreviewPmemPool : public PmemPool {
      virtual ~PreviewPmemPool();
      PreviewPmemPool(int buffer_size, int num_buffers,
                         int frame_size,
                         int frame_offset,
                         const char *name);
};
struct RawPmemPool : public PmemPool {
      virtual ~RawPmemPool();
       RawPmemPool(const char *pmem_pool,
                     int buffer_size, int num_buffers,
                     int frame_size,
                     int frame_offset,
                     const char *name);
};
```

21.4　实战演练——在 OMAP 平台实现 Camera 驱动

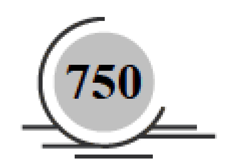
在 OMAP 平台中，可以使用高级的 ISP（图像信号处理）模块通过外接（I2C 方式连接）的 Camera Sensor 驱动来获取视频帧的数据。在 OMAP 平台中，和 Camera 系统相关的实现文件保存在目录 drivers/media/video/中。

在此目录中，主要由如下 3 部分组成。

- ☑ Vedio for Linux 2 设备：实现文件是 omap34xxcam.h 和 omap34xxcam.c。
- ☑ ISP：实现文件是 isp 目录中的 isp.c、isph3a.c、isppreview.c、ispresizer.c，提供了通过 ISP 进行的 3A、预览、改变尺寸等功能。
- ☑ Camera Sensor 驱动：lv8093.c 或 imx046.c，使用 v4l2-int-device 结构来注册。

在文件 omap34xxcam.c 中通过 v4l2_int_master 定义了 v4l2_int 主设备，对应的代码如下。

```
static struct v4l2_int_master omap34xxcam_master = {
           .attach = omap34xxcam_device_register,                //注册设备
```



```
        .detach = omap34xxcam_device_unregister,          //注销设备
};
```

还需要定义 omap34xxcam_fops 来注册 video 中的 v4l2_file_operations 结构，定义代码如下。

```
static struct v4l2_file_operations omap34xxcam_fops = {
        .owner = THIS_MODULE,
        .unlocked_ioctl = video_ioctl2,
        .poll = omap34xxcam_poll,
        .mmap = omap34xxcam_mmap,
        .open = omap34xxcam_open,
        .release = omap34xxcam_release,
};
```

另外还需要通过文件 lv8093.c 或 imx046.c 实现 Camera 系统的传感器功能，并连接在系统的 I2C 总线上。通过结构 v4l2-int-device 从设备进行注册，在运行时被文件 omap34xxcam.c 直接调用。

OMAP 平台的 Camera 硬件抽象层是基于 OMAP 的 V4L2 驱动程序实现的，并调用 Overlay 系统作为视频输出，所以 Camera 硬件抽象层的 useOverlay()的返回值是 true。为了提高性能，需要直接映射 Overlay 中的内存以作为 Camera 输出的内存。当在 OMAP 的 Camera 硬件抽象层中调用 V4L2 驱动程序时，需要使用 V4L2_MEMORY_USERPTR 标识来表示来自用户空间的内存。

在 OMAP 平台的 Camera 硬件抽象层中可以使用自动对焦 AutoFocus、自动增强 AutoEnhance 和自动平衡 AutoWhiteBalance 等增强型功能。上述增强型功能是通过 OMAP SOC 内部的 ISP 模块提供的基本机制实现的，算法部分功能是由用户空间库所支持的。

21.5　Android 实现 S5PV210 FIMC 驱动

本书中讲解的 FIMC 是 Samsung Camera Interface Driver 的缩写，是三星公司的一款摄像头驱动。FIMC 的实现文件是 fimc_capture.c，在 FIMC 系统中的位置如图 21-3 所示。

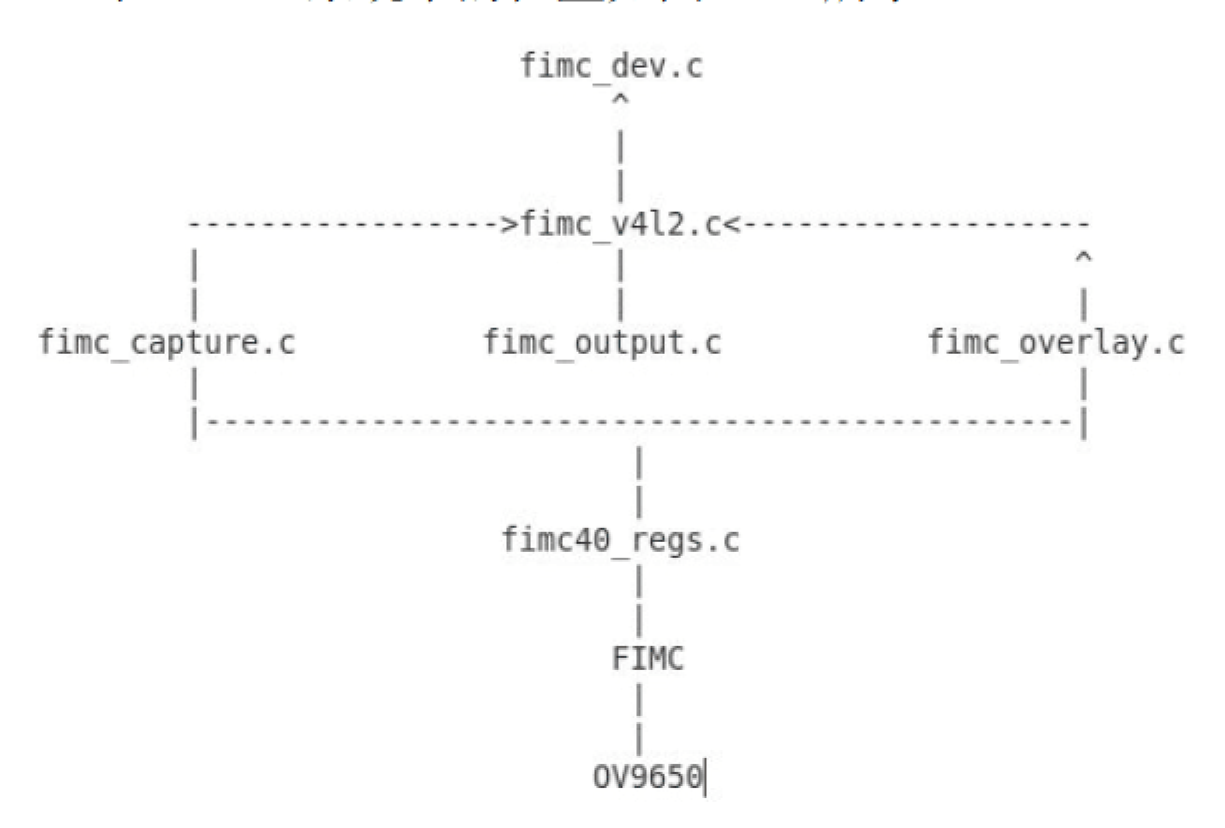


图 21-3　FIMC 系统的结构

文件 fimc_regs.c 是 FIMC 框架操作 Camera 硬件的接口，FIMC 框架把所有硬件相关的操作都放在这个文件中。文件 fimc_regs.c 的主要实现代码如下。

```
int fimc_hwset_camera_source(struct fimc_control *ctrl)
{
    struct s3c_platform_camera *cam = ctrl->cam;
    u32 cfg = 0;
```

```
104
105      /* for now, we support only ITU601 8 bit mode */
106      cfg |= S3C_CISRCFMT_ITU601_8BIT;
107      cfg |= cam->order422;
108
109      if (cam->type == CAM_TYPE_ITU)
110          cfg |= cam->fmt;
111
112      cfg |= S3C_CISRCFMT_SOURCEHSIZE(cam->width);
113      cfg |= S3C_CISRCFMT_SOURCEVSIZE(cam->height);
114
115      writel(cfg, ctrl->regs + S3C_CISRCFMT);
116
117      return 0;
118  }
```

在上述代码中，S3C_CISRCFMT、Camera Source Format 和 FIMC1 FIMC2 FIMC3 各对应一个 external 摄像头支持的模式。一般来说，AD 转换芯片都支持 BT656。在第 107 行代码中，cam->order422 中的 cam 代表一个外部摄像头，cam->order422 是在文件 arch/arm/mach-s5pv210/mach-xxx.c 中定义的，标识了 external camera 像素的 YCR 分量的排列方式。对于 BT656 来说，可以选择如下选项：

- ☑ CAM_ORDER422_8BIT_YCBYCR
- ☑ CAM_ORDER422_8BIT_YCRYCB
- ☑ CAM_ORDER422_8BIT_CBYCRY
- ☑ CAM_ORDER422_8BIT_CRYCBY

具体选择哪一个，要根据 sensor datasheet 中 BT656 输出 YUV 分量的顺序决定。再看第 109 行代码，因为 cam->fmt 也是设置 ITU 模式的，所以和第 106 行代码是冗余的。第 112、113 行代码用于设置 source 水平和垂直像素数目，source 可以是 camera 或者 FIFO input。

再看下面的代码，其中 CITAREA 用于设置 output DMA 的 target 大小，这个值并不是 Buffer 空间的大小，而是输出图像的 H_size×V_size。

```
int fimc_hwset_output_area_size(struct fimc_control *ctrl, u32 size)
{
    u32 cfg = 0;

    cfg = S3C_CITAREA_TARGET_AREA(size);

    writel(cfg, ctrl->regs + S3C_CITAREA);

    return 0;
}
```

再看下面的代码，在 disable capture 后可以调用下面这个函数，以保证 disable capture 操作完成。其中 S3C_CISTATUS_IMGCPTEN 用于标识是否 image capture enable 的状态。

```
void fimc_wait_disable_capture(struct fimc_control *ctrl)
{
    unsigned long timeo = jiffies + 20; /* timeout of 100 ms */
    u32 cfg;

    if (!ctrl || !ctrl->cap ||
            ctrl->cap->fmt.colorspace == V4L2_COLORSPACE_JPEG)
        return;
```

```

    while (time_before(jiffies, timeo)) {
        cfg = readl(ctrl->regs + S3C_CISTATUS);

        if (0 == (cfg & S3C_CISTATUS_IMGCPTEN))
            break;

        msleep(10);
    }

    dev_dbg(ctrl->dev, "IMGCPTEN: Wait time = %d ms\n",
            jiffies_to_msecs(jiffies - timeo + 20));

    return;
}
```

再看下面的代码，因为 FIMC 控制器支持图片特效处理，所以 fimc 的 V4L2 s_ctl 接口提供了特效控制。其中 CIIMGEFF 寄存器用于控制图片的特效，具体的特效说明可参考 S5PV210 datasheet。

```
int fimc_hwset_image_effect(struct fimc_control *ctrl)
{
    u32 cfg = 0;

    if (ctrl->fe.ie_on) {
        if (ctrl->fe.ie_after_sc)
            cfg |= S3C_CIIMGEFF_IE_SC_AFTER;

        cfg |= S3C_CIIMGEFF_FIN(ctrl->fe.fin);

        if (ctrl->fe.fin == FIMC_EFFECT_FIN_ARBITRARY_CBCR)
            cfg |= S3C_CIIMGEFF_PAT_CB(ctrl->fe.pat_cb) |
                S3C_CIIMGEFF_PAT_CR(ctrl->fe.pat_cr);

        cfg |= S3C_CIIMGEFF_IE_ENABLE;
    }

    writel(cfg, ctrl->regs + S3C_CIIMGEFF);

    return 0;
}
```

然后通过函数 fimc_hwset_reset()实现软件复位处理，具体代码如下所示。

```
int fimc_hwset_reset(struct fimc_control *ctrl)
{
u32 cfg = 0;

cfg = readl(ctrl->regs + S3C_CISRCFMT);
cfg |= S3C_CISRCFMT_ITU601_8BIT;
writel(cfg, ctrl->regs + S3C_CISRCFMT);

/* s/w reset */
cfg = readl(ctrl->regs + S3C_CIGCTRL);
cfg |= (S3C_CIGCTRL_SWRST);
```

```
writel(cfg, ctrl->regs + S3C_CIGCTRL);
mdelay(1);

cfg = readl(ctrl->regs + S3C_CIGCTRL);
cfg &= ~S3C_CIGCTRL_SWRST;
writel(cfg, ctrl->regs + S3C_CIGCTRL);

/* in case of ITU656, CISRCFMT[31] should be 0 */
if ((ctrl->cap != NULL) && (ctrl->cam->fmt == ITU_656_YCBCR422_8BIT)) {
cfg = readl(ctrl->regs + S3C_CISRCFMT);
cfg &= ~S3C_CISRCFMT_ITU601_8BIT;
writel(cfg, ctrl->regs + S3C_CISRCFMT);
}

fimc_reset_cfg(ctrl);

return 0;
}
```

在上述复位过程中，S5PV210 datasheet 推荐使用如下所示的初始化序列。

☑ 对于 ITU601：ITU601_656n 置 1 -> SwRst 置 1 -> SwRst 置 0。

☑ 对于 ITU656：ITU601_656n 置 1 -> SwRst 置 1 -> SwRst 置 0 -> ITU601_656 置 0。

再看如下所示的函数 fimc_hwset_camera_offset()，功能是实现坐标定位处理。

```
int fimc_hwset_camera_offset(struct fimc_control *ctrl)
{
    struct s3c_platform_camera *cam = ctrl->cam;
    struct v4l2_rect *rect = &cam->window;
    u32 cfg, h1, h2, v1, v2;

    if (!cam) {
        fimc_err("%s: no active camera\n", __func__);
        return -ENODEV;
    }

    h1 = rect->left;
    h2 = cam->width - rect->width - rect->left;
    v1 = rect->top;
    v2 = cam->height - rect->height - rect->top;

    cfg = readl(ctrl->regs + S3C_CIWDOFST);
    cfg &= ~(S3C_CIWDOFST_WINHOROFST_MASK | S3C_CIWDOFST_WINVEROFST_MASK);
    cfg |= S3C_CIWDOFST_WINHOROFST(h1);
    cfg |= S3C_CIWDOFST_WINVEROFST(v1);
    cfg |= S3C_CIWDOFST_WINOFSEN;
    writel(cfg, ctrl->regs + S3C_CIWDOFST);

    cfg = 0;
    cfg |= S3C_CIWDOFST2_WINHOROFST2(h2);
    cfg |= S3C_CIWDOFST2_WINVEROFST2(v2);
    writel(cfg, ctrl->regs + S3C_CIWDOFST2);

```

```
    return 0;
}
```

接下来分析文件 fimc_capture.c，首先看如下所示的代码。

```
static const struct v4l2_fmtdesc capture_fmts[] = {
    {
        .index = 0,
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .flags = FORMAT_FLAGS_PACKED,
        .description = "RGB-5-6-5",
        .pixelformat = V4L2_PIX_FMT_RGB565,
    }, {
        .index = 1,
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .flags = FORMAT_FLAGS_PACKED,
        .description = "RGB-8-8-8, unpacked 24 bpp",
        .pixelformat = V4L2_PIX_FMT_RGB32,
    }, {
        .index = 2,
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .flags = FORMAT_FLAGS_PACKED,
        .description = "YUV 4:2:2 packed, YCbYCr",
        .pixelformat = V4L2_PIX_FMT_YUYV,
    }, {
        .index = 3,
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .flags = FORMAT_FLAGS_PACKED,
        .description = "YUV 4:2:2 packed, CbYCrY",
        .pixelformat = V4L2_PIX_FMT_UYVY,
    }, {
        .index = 4,
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .flags = FORMAT_FLAGS_PACKED,
        .description = "YUV 4:2:2 packed, CrYCbY",
        .pixelformat = V4L2_PIX_FMT_VYUY,
    }, {
        .index = 5,
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .flags = FORMAT_FLAGS_PACKED,
        .description = "YUV 4:2:2 packed, YCrYCb",
        .pixelformat = V4L2_PIX_FMT_YVYU,
    }, {
        .index = 6,
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .flags = FORMAT_FLAGS_PLANAR,
        .description = "YUV 4:2:2 planar, Y/Cb/Cr",
        .pixelformat = V4L2_PIX_FMT_YUV422P,
    }, {
        .index = 7,
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .flags = FORMAT_FLAGS_PLANAR,
        .description = "YUV 4:2:0 planar, Y/CbCr",
```

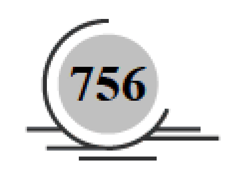
```
            .pixelformat = V4L2_PIX_FMT_NV12,
    }, {
            .index = 8,
            .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
            .flags = FORMAT_FLAGS_PLANAR,
            .description = "YUV 4:2:0 planar, Y/CbCr, Tiled",
            .pixelformat = V4L2_PIX_FMT_NV12T,
    }, {
            .index = 9,
            .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
            .flags = FORMAT_FLAGS_PLANAR,
            .description = "YUV 4:2:0 planar, Y/CrCb",
            .pixelformat = V4L2_PIX_FMT_NV21,
    }, {
            .index = 10,
            .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
            .flags = FORMAT_FLAGS_PLANAR,
            .description = "YUV 4:2:2 planar, Y/CbCr",
            .pixelformat = V4L2_PIX_FMT_NV16,
    }, {
            .index = 11,
            .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
            .flags = FORMAT_FLAGS_PLANAR,
            .description = "YUV 4:2:2 planar, Y/CrCb",
            .pixelformat = V4L2_PIX_FMT_NV61,
    }, {
            .index = 12,
            .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
            .flags = FORMAT_FLAGS_PLANAR,
            .description = "YUV 4:2:0 planar, Y/Cb/Cr",
            .pixelformat = V4L2_PIX_FMT_YUV420,
    }, {
            .index = 13,
            .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
            .flags = FORMAT_FLAGS_ENCODED,
            .description = "Encoded JPEG bitstream",
            .pixelformat = V4L2_PIX_FMT_JPEG,
    },
};
```

在上述代码列表中列出了 FIMC 支持的 capture 格式，应用程序可以通过 vidioc_s_fmt 设置 capture 的输出格式，capture 的输出格式必须在上面的列表中。这里的 flags 标志位并不符合 V4L2 标准，V4L2 只支持一种标志，即 V4L2_FMT_FLAG_COMPRESSED。三星扩展了 flags 标志，具体说明如下所示。

- ☑ FORMAT_FLAGS_PACKED：图片的像素点分量放在同一个 Buffer 中。
- ☑ FORMAT_FLAGS_PLANAR：图片像素的分量放在不同的 Buffer 中。
- ☑ FORMAT_FLAGS_ENCODED：图片数据编码存储，如 jpeg 格式。

在下面的代码中定义了 FIMC 支持 ctrl，其中以下 4 个 ctrl 是 Samsung FIMC 私有的 ctrl id，分别用来获取分量的物理起始地址。

- ☑ V4L2_CID_PADDR_Y
- ☑ V4L2_CID_PADDR_CB



☑ V4L2_CID_PADDR_CR

☑ V4L2_CID_PADDR_CBCR

```
static int fimc_camera_init(struct fimc_control *ctrl)
{
    int ret;

    fimc_dbg("%s\n", __func__);

    /* do nothing if already initialized */
    if (ctrl->cam->initialized)
        return 0;

    /* enable camera power if needed */
    if (ctrl->cam->cam_power)
        ctrl->cam->cam_power(1);

    /* subdev call for init */
    ret = subdev_call(ctrl, core, init, 0);
    if (ret == -ENOIOCTLCMD) {
        fimc_err("%s: init subdev api not supported\n",
            __func__);
        return ret;
    }

    if (ctrl->cam->type == CAM_TYPE_MIPI) {
        /* subdev call for sleep/wakeup:
         * no error although no s_stream api support
         */
        u32 pixelformat;
        if (ctrl->cap->fmt.pixelformat == V4L2_PIX_FMT_JPEG)
            pixelformat = V4L2_PIX_FMT_JPEG;
        else
            pixelformat = ctrl->cam->pixelformat;

        subdev_call(ctrl, video, s_stream, 0);
        s3c_csis_start(ctrl->cam->mipi_lanes, ctrl->cam->mipi_settle, \
                ctrl->cam->mipi_align, ctrl->cam->width, \
                ctrl->cam->height, pixelformat);
        subdev_call(ctrl, video, s_stream, 1);
    }

    ctrl->cam->initialized = 1;

    return 0;
}
```

上述函数的主要功能是对 Camera 的 sensor 进行上电和初始化操作，上述函数最早的调用位置是streamon。但是此处有一个问题，假定外围电路是一个 video AD 转换芯片，或多个 cvbs s-video，或者 YPbPr 输入，那么在执行 streamon 之前需要先执行 s_input 操作选择的那个 video AD 芯片的输入。选择 video AD 的 input 输入是要操作 AD 芯片 I2C 寄存器的，因此这个上电位置是有问题的。

```
static int fimc_add_inqueue(struct fimc_control *ctrl, int i)
{
```

```
    struct fimc_capinfo *cap = ctrl->cap;

    struct fimc_buf_set *buf;

    if (i >= cap->nr_bufs)
        return -EINVAL;

    list_for_each_entry(buf, &cap->inq, list) {
        if (buf->id == i) {
            fimc_dbg("%s: buffer %d already in inqueue.\n", \
                    __func__, i);
            return -EINVAL;
        }
    }

    list_add_tail(&cap->bufs[i].list, &cap->inq);

    return 0;
}
```

上述函数被 qbuf 调用，把@i 指定的 Buffer 加到 cap->inq 链表中。cap->inq 是可用 Buffer 链表，当 FIMC 更新 out DMA address 时，就设置为 cap->inq 中的一个 Buffer。

```
static int fimc_add_outqueue(struct fimc_control *ctrl, int i)
{
    struct fimc_capinfo *cap = ctrl->cap;
    struct fimc_buf_set *buf;

    unsigned int mask = 0x2;

    /* PINGPONG_2ADDR_MODE Only */
    /* pair_buf_index stands for pair index of i. (0<->2) (1<->3) */

    int pair_buf_index = (i^mask);

    /* FIMC have 4 h/w registers */
    if (i < 0 || i >= FIMC_PHYBUFS) {
        fimc_err("%s: invalid queue index : %d\n", __func__, i);
        return -ENOENT;
    }

    if (list_empty(&cap->inq))
        return -ENOENT;

    buf = list_first_entry(&cap->inq, struct fimc_buf_set, list);

    /* pair index buffer should be allocated first */
    cap->outq[pair_buf_index] = buf->id;
    fimc_hwset_output_address(ctrl, buf, pair_buf_index);

    cap->outq[i] = buf->id;
    fimc_hwset_output_address(ctrl, buf, i);

```

```
420     if (cap->nr_bufs != 1)
421         list_del(&buf->list);
422
423     return 0;
424 }
```

对上述代码的具体说明如下。

☑ 411 行：在 cap->inq Buffer 链表中取得第一个可用 Buffer。

☑ 413～418 行：把 buf 设置到两个输出 out DMA address 寄存器中，最多可以把 4 个 out DMA address 都配置上，这样可以增加画面的流畅度。4 个 output DMA address 把帧数分成 4 个部分，第 1 个 DMA address 存储 1、5、9、13…帧；第 2 个 DMA address 存储 2、6、10、14…帧；第 3 个存储 3、7、11、15…帧；第 4 个存储 4、8、12、16...帧，如果仅使用一个 output DMA address，那么仅能得到 1/4 的帧率。

☑ 420 和 421 行：从 cap->inq 链表中删除这个 Buffer。

通过函数 fimc_reqbufs_capture()分配视频输入内存，具体实现代码如下。

```
int fimc_reqbufs_capture(void *fh, struct v4l2_requestbuffers *b)
{
    struct fimc_control *ctrl = ((struct fimc_prv_data *)fh)->ctrl;
    struct fimc_capinfo *cap = ctrl->cap;
    int ret = 0, i;
    int size[4] = { 0, 0, 0, 0};
    int align = SZ_4K;

    if (b->memory != V4L2_MEMORY_MMAP) {
        fimc_err("%s: invalid memory type\n", __func__);
        return -EINVAL;
    }

    if (!cap) {
        fimc_err("%s: no capture device info\n", __func__);
        return -ENODEV;
    }

    if (!ctrl->cam || !ctrl->cam->sd) {
        fimc_err("%s: No capture device.\n", __func__);
        return -ENODEV;
    }

    mutex_lock(&ctrl->v4l2_lock);

    if (b->count < 1 || b->count > FIMC_CAPBUFS)
        return -EINVAL;

    /* It causes flickering as buf_0 and buf_3 refer to same hardware
     * address.
     */
    if (b->count == 3)
        b->count = 4;

    cap->nr_bufs = b->count;

```

```
	fimc_dbg("%s: requested %d buffers\n", __func__, b->count);

	INIT_LIST_HEAD(&cap->inq);

	fimc_free_buffers(ctrl);

	switch (cap->fmt.pixelformat) {
	case V4L2_PIX_FMT_RGB32:	/* fall through */
	case V4L2_PIX_FMT_RGB565:	/* fall through */
	case V4L2_PIX_FMT_YUYV:	/* fall through */
	case V4L2_PIX_FMT_UYVY:	/* fall through */
	case V4L2_PIX_FMT_VYUY:	/* fall through */
	case V4L2_PIX_FMT_YVYU:	/* fall through */
	case V4L2_PIX_FMT_YUV422P:	/* fall through */
		size[0] = cap->fmt.sizeimage;
		break;

	case V4L2_PIX_FMT_NV16:		/* fall through */
	case V4L2_PIX_FMT_NV61:
		size[0] = cap->fmt.width * cap->fmt.height;
		size[1] = cap->fmt.width * cap->fmt.height;
		size[3] = 16; /* Padding buffer */
		break;
	case V4L2_PIX_FMT_NV12:
		size[0] = cap->fmt.width * cap->fmt.height;
		size[1] = cap->fmt.width * cap->fmt.height/2;
		break;
	case V4L2_PIX_FMT_NV21:
		size[0] = cap->fmt.width * cap->fmt.height;
		size[1] = cap->fmt.width * cap->fmt.height/2;
		size[3] = 16; /* Padding buffer */
		break;
	case V4L2_PIX_FMT_NV12T:
		/* Tiled frame size calculations as per 4x2 tiles
		 *	- Width: Has to be aligned to 2 times the tile width
		 *	- Height: Has to be aligned to the tile height
		 *	- Alignment: Has to be aligned to the size of the
		 *	macrotile (size of 4 tiles)
		 *
		 * NOTE: In case of rotation, we need modified calculation as
		 * width and height are aligned to different values.
		 */
		if (cap->rotate == 90 || cap->rotate == 270) {
			size[0] = ALIGN(ALIGN(cap->fmt.height, 128) *
					ALIGN(cap->fmt.width, 32),
					SZ_8K);
			size[1] = ALIGN(ALIGN(cap->fmt.height, 128) *
					ALIGN(cap->fmt.width/2, 32),
					SZ_8K);
		} else {
			size[0] = ALIGN(ALIGN(cap->fmt.width, 128) *
					ALIGN(cap->fmt.height, 32),
					SZ_8K);
```

```
1039                size[1] = ALIGN(ALIGN(cap->fmt.width, 128) *
1040                        ALIGN(cap->fmt.height/2, 32),
1041                        SZ_8K);
1042            }
1043            align = SZ_8K;
1044            break;
1045
1046        case V4L2_PIX_FMT_YUV420:
1047            size[0] = cap->fmt.width * cap->fmt.height;
1048            size[1] = cap->fmt.width * cap->fmt.height >> 2;
1049            size[2] = cap->fmt.width * cap->fmt.height >> 2;
1050            size[3] = 16; /* Padding buffer */
1051            break;
1052
1053        case V4L2_PIX_FMT_JPEG:
1054            size[0] = fimc_camera_get_jpeg_memsize(ctrl);
1055        default:
1056            break;
1057        }
1058
1059        ret = fimc_alloc_buffers(ctrl, size, align);
1060        if (ret) {
1061            fimc_err("%s: no memory for "
1062                    "capture buffer\n", __func__);
1063            mutex_unlock(&ctrl->v4l2_lock);
1064            return -ENOMEM;
1065        }
1066
1067        for (i = cap->nr_bufs; i < FIMC_PHYBUFS; i++) {
1068            memcpy(&cap->bufs[i], \
1069                &cap->bufs[i - cap->nr_bufs], sizeof(cap->bufs[i]));
1070        }
1071
1072        mutex_unlock(&ctrl->v4l2_lock);
1073
1074        return 0;
1075    }
```

对上述代码的具体说明如下。

- ☑ 975～978 行：FIMC_CAPBUFS 是 FIMC 支持的最大 queue buffers 数量，可以根据最大 capture buffers 数目，以及帧 Buffer 所需空间大小（所有子 Buffers 空间总和），加上 alignment 所带来的空间损失，大致算出 fimc capture 设备需要预留的物理空间。
- ☑ 992～1057 行：根据 pixelformat 和 width/height 计算每个帧子 Buffers 的尺寸。

函数 fimc_cropcap_capture()是 FIMC 的 VIDIOC_CROPCAP 实现，具体实现代码如下。

```
1255    int fimc_cropcap_capture(void *fh, struct v4l2_cropcap *a)
1256    {
1257        struct fimc_control *ctrl = ((struct fimc_prv_data *)fh)->ctrl;
1258        struct fimc_capinfo *cap = ctrl->cap;
1259
1260        fimc_dbg("%s\n", __func__);
1261
1262        if (!ctrl->cam || !ctrl->cam->sd || !ctrl->cap) {
```

```
            fimc_err("%s: No capture device.\n", __func__);
            return -ENODEV;
        }

        mutex_lock(&ctrl->v4l2_lock);

        /* crop limitations */
        cap->cropcap.bounds.left = 0;
        cap->cropcap.bounds.top = 0;
        cap->cropcap.bounds.width = ctrl->cam->width;
        cap->cropcap.bounds.height = ctrl->cam->height;

        /* crop default values */
        cap->cropcap.defrect.left = 0;
        cap->cropcap.defrect.top = 0;
        cap->cropcap.defrect.width = ctrl->cam->width;
        cap->cropcap.defrect.height = ctrl->cam->height;

        a->bounds = cap->cropcap.bounds;
        a->defrect = cap->cropcap.defrect;

        mutex_unlock(&ctrl->v4l2_lock);

        return 0;
    }
```

cropcap.bounds 是 capture window 最大边界，capture.defrect 是 capture window 的默认方框，cropcap. defrect 一定不会超出 cropcap.bounds 的范围，它们的关系如图 21-4 所示。

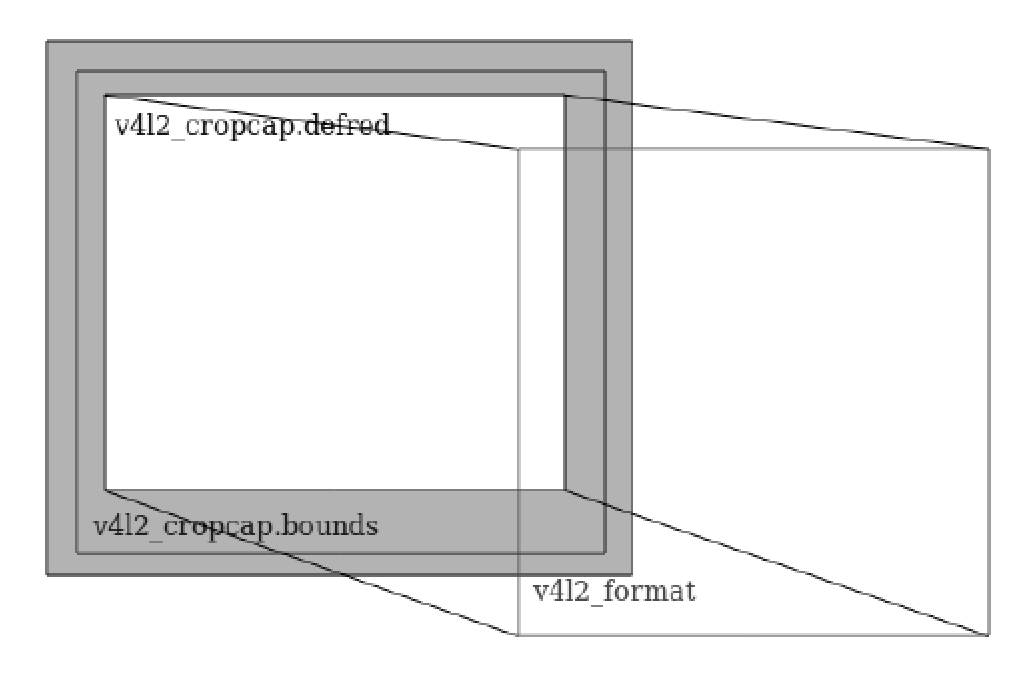


图 21-4　cropcap.bounds 和 capture.defrect 关系图

其中 cropcap.pixelaspect＝垂直像素数/水平像素数，而函数 fimc_g_crop_capture()是 capture 设备的 VIDIOC_G_CROP 实现，功能是返回当前的 crop，具体实现代码如下。

```
static int fimc_capture_crop_size_check(struct fimc_control *ctrl)
{
        struct fimc_capinfo *cap = ctrl->cap;
        int win_hor_offset = 0, win_hor_offset2 = 0;
        int win_ver_offset = 0, win_ver_offset2 = 0;
        int crop_width = 0, crop_height = 0;

        /* check win_hor_offset, win_hor_offset2 */
        win_hor_offset = ctrl->cam->window.left;
```

```
	win_hor_offset2 = ctrl->cam->width - ctrl->cam->window.left -
					ctrl->cam->window.width;

	win_ver_offset = ctrl->cam->window.top;
	win_ver_offset2 = ctrl->cam->height - ctrl->cam->window.top -
					ctrl->cam->window.height;

	if (win_hor_offset < 0 || win_hor_offset2 < 0) {
		fimc_err("%s: Offset (left-side(%d) or right-side(%d) "
				"is negative.\n", __func__, \
				win_hor_offset, win_hor_offset2);
		return -1;
	}

	if (win_ver_offset < 0 || win_ver_offset2 < 0) {
		fimc_err("%s: Offset (top-side(%d) or bottom-side(%d)) "
				"is negative.\n", __func__, \
				win_ver_offset, win_ver_offset2);
		return -1;
	}

	if ((win_hor_offset % 2) || (win_hor_offset2 % 2)) {
		fimc_err("%s: win_hor_offset must be multiple of 2\n", \
				__func__);
		return -1;
	}

	/* check crop_width, crop_height */
	crop_width = ctrl->cam->window.width;
	crop_height = ctrl->cam->window.height;

	if (crop_width % 16) {
		fimc_err("%s: crop_width must be multiple of 16\n", __func__);
		return -1;
	}

	switch (cap->fmt.pixelformat) {
	case V4L2_PIX_FMT_YUV420:		/* fall through */
	case V4L2_PIX_FMT_NV12:			/* fall through */
	case V4L2_PIX_FMT_NV21:			/* fall through */
	case V4L2_PIX_FMT_NV12T:		/* fall through */
		if ((crop_height % 2) || (crop_height < 8)) {
			fimc_err("%s: crop_height error!\n", __func__);
			return -1;
		}
		break;
	default:
		break;
	}

	return 0;
}
```

第 22 章　蓝牙系统驱动

蓝牙是一种支持设备短距离通信（一般 10 米内）的无线电技术，可以在包括移动电话、PDA、无线耳机、笔记本电脑、相关外设等众多设备之间进行无线信息交换。本章将首先讲解 Android 5.0 系统中蓝牙模块的底层源码和实现原理，为读者学习本书后面的知识打下基础。

22.1　Android 系统中的蓝牙模块

Android 系统包含了对蓝牙网络协议栈的支持，这使得蓝牙设备能够无线连接其他蓝牙设备交换数据。Android 的应用程序框架提供了访问蓝牙功能的 APIs。这些 APIs 让应用程序能够无线连接其他蓝牙设备，实现点对点或点对多点的无线交互功能。

通过使用蓝牙 APIs，一个 Android 应用程序能够实现如下功能。

☑　扫描其他蓝牙设备。
☑　查询本地蓝牙适配器（local Bluetooth adapter）用于配对蓝牙设备。
☑　建立 RFCOMM 信道（channels）。
☑　通过服务发现（service discovery）连接其他设备。
☑　数据通信。
☑　管理多个连接。

Android 平台中的蓝牙系统是基于 BlueZ 实现的，BlueZ 是通过 Linux 中的一套完整的蓝牙协议栈开源实现的。当前 BlueZ 被广泛应用于各种 Linux 版本中，并被芯片公司移植到各种芯片平台上使用。在 Linux 2.6 内核中已经包含了完整的 BlueZ 协议栈，在 Android 系统中已经移植并嵌入进了 BlueZ 的用户空间实现，并且随着硬件技术的发展而不断更新。

蓝牙（Bluetooth）技术实际上是一种短距离无线电技术。在 Android 系统的蓝牙模块中，除了使用 Kernel 支持外，还需要用户空间的 BlueZ 支持。

Android 平台中蓝牙模块的基本层次结构如图 22-1 所示。

在 Android 平台中，蓝牙系统从上到下主要包括 Java 框架中的 Bluetooth 类、Android 适配库、BlueZ 库、驱动程序和协议，这几部分的具体结构如图 22-2 所示。

在图 22-2 中各个层次结构的具体说明如下。

（1）BlueZ 库

Android 蓝牙设备管理库的路径是 external/bluez/。

可以分别生成库 libbluetooth.so、libbluedroid.so 和 hcidump 等众多相关工具和库。BlueZ 库提供了对用户空间蓝牙的支持，在里面包含了主机控制协议 HCI 以及其他众多内核实现协议的接口，并且实现了所有蓝牙应用模式 Profile。

（2）蓝牙的 JNI 部分

此部分的代码路径是 frameworks/base/core/jni/。

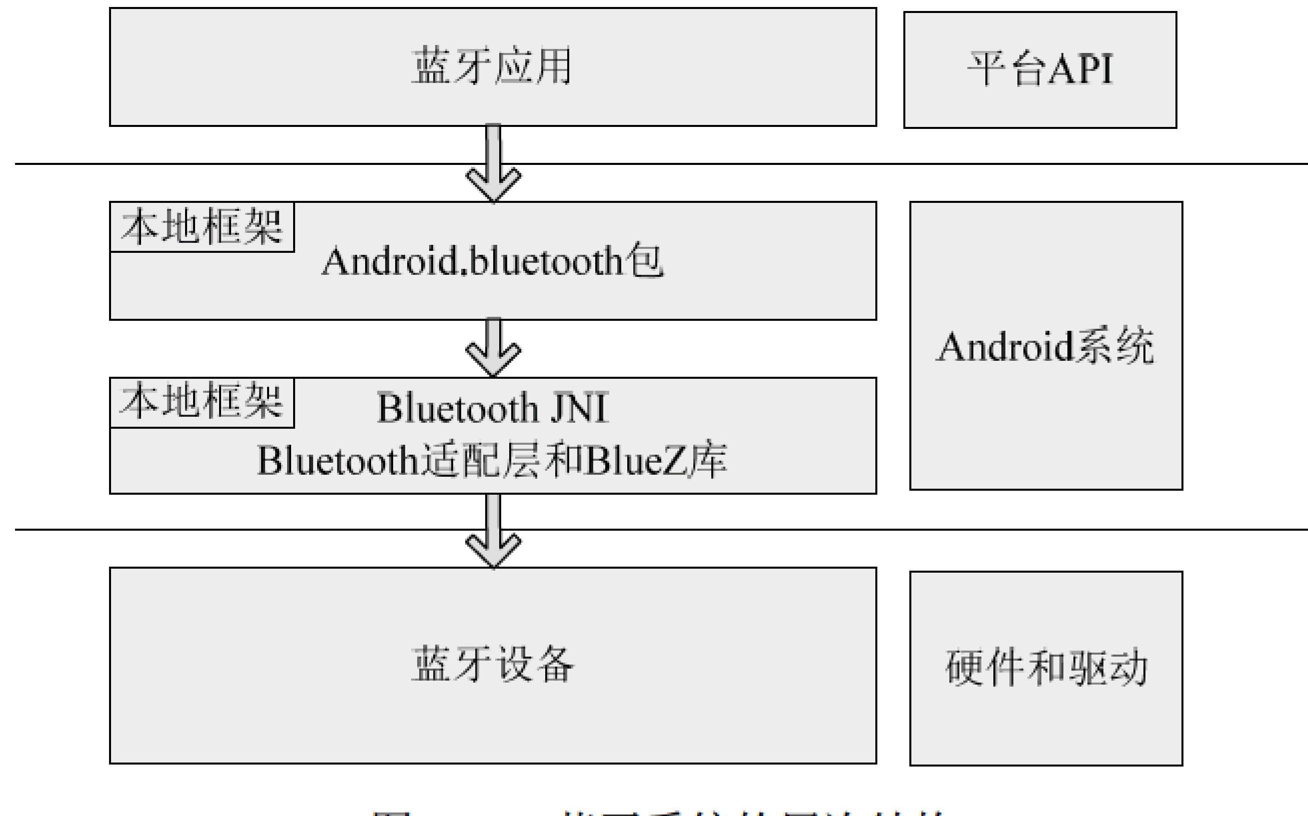


图 22-1　蓝牙系统的层次结构

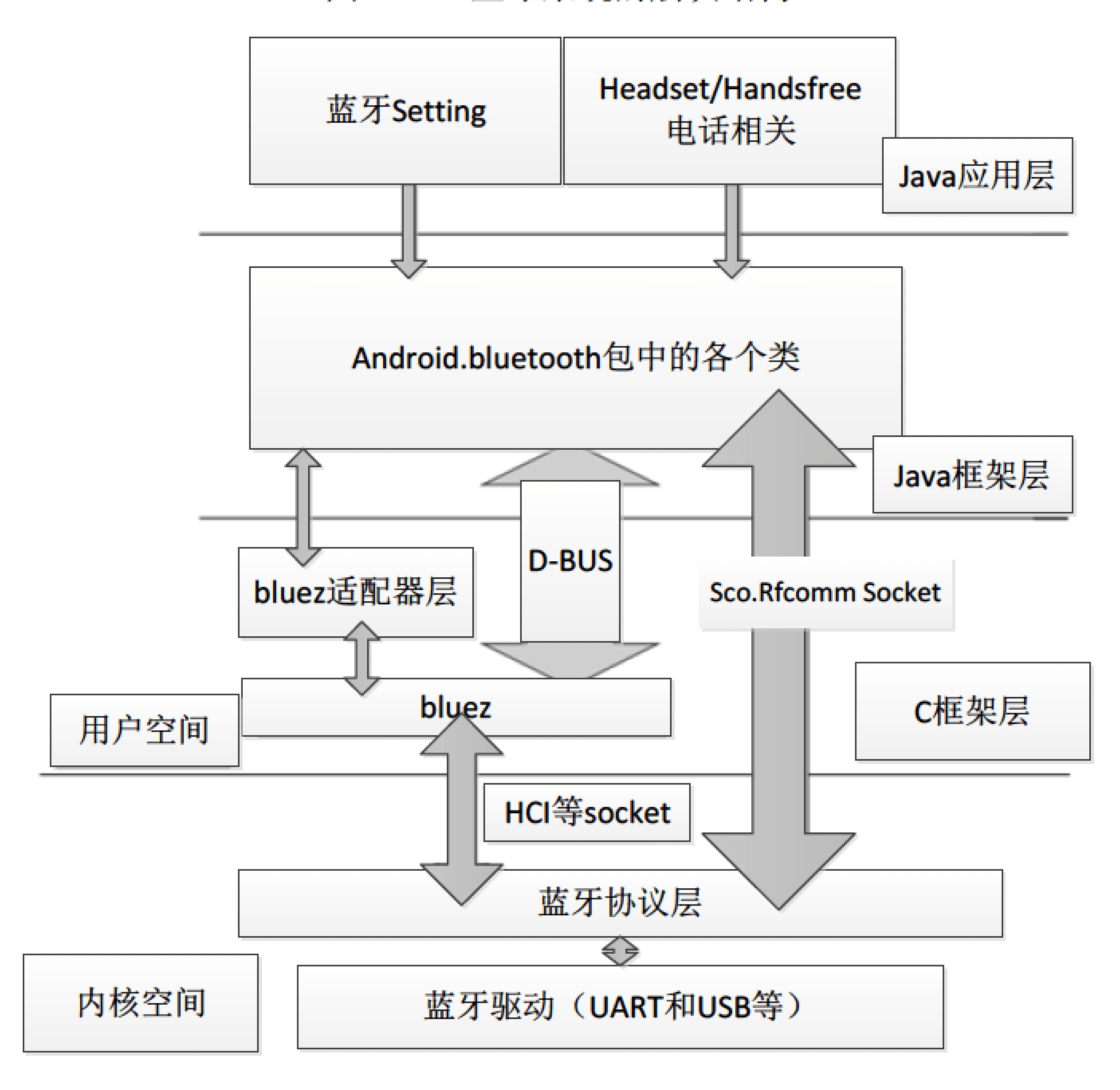


图 22-2　蓝牙系统结构

（3）Java 框架层

此部分的代码路径如下所示。

☑　frameworks/base/core/java/android/bluetooth：蓝牙部分对应应用程序的 API。

☑　frameworks/base/core/java/android/Server：蓝牙的服务部分。

蓝牙的服务部分负责管理并使用底层本地服务，并封装成系统服务，而在 android.bluetooth 部分中包含了各个蓝牙平台的 API 部分，以供应用程序层所使用。

（4）Bluetooth 的适配库

此部分的代码路径是 system/bluetooth/。

在此层用于生成库 libbluedroid.so 以及相关工具和库，能够实现对蓝牙设备的管理，例如蓝牙设备的电源管理。

22.2 分析蓝牙模块的源码

要想完全掌握蓝牙系统的开发原理，需要首先分析 Android 中的蓝牙源码并了解其核心构造，只有这样才能对蓝牙应用开发做到游刃有余。本节将简要介绍开源 Android 中蓝牙模块的相关代码。

22.2.1 初始化蓝牙芯片

初始化蓝牙芯片工作是通过 BlueZ 工具 hciattach 进行的，此工具在目录 external/bluetooth/tools 的文件中实现。

hciattach 命令主要用来初始化蓝牙设备，其命令格式如下。

```
hciattach [-n] [-p] [-b] [-t timeout] [-s initial_speed] <tty> <type | id> [speed] [flow|noflow] [bdaddr]
```

在上述格式中，其中最重要的参数就是 type 和 speed，type 决定了要初始化的设备的型号，可以使用“hciattach -l”列出所支持的设备型号。

并不是所有的参数对所有的设备都是适用的，有些设备会忽略一些参数设置，例如，查看 hciattach 的代码就可以看到，多数设备都忽略 bdaddr 参数。hciattach 命令内部的工作步骤是：首先打开制定的 tty 设备，然后做一些通用的设置，如 flow 等，然后设置波特率为 initial_speed，然后根据 type 调用各自的初始化代码，最后将波特率重新设置为 speed。所以调用 hciattach 时，要根据实际情况设置好 initial_speed 和 speed。

对于 type BCSP 来说，它的初始化代码只做了一件事，就是完成 BCSP 协议的同步操作，它并不对蓝牙芯片做任何的 pskey 设置。

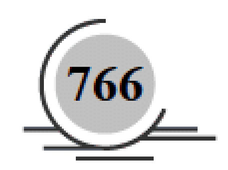
22.2.2 蓝牙服务

一般不需要我们自己定义蓝牙方面的服务，只需要使用初始化脚本文件 init.rc 中的默认内容即可。例如下面的代码。

```
service bluetoothd /system/bin/logwrapper /system/bin/bluetoothd -d -n
    socket bluetooth stream 660 bluetooth bluetooth
    socket dbus_bluetooth stream 660 bluetooth bluetooth
    # init.rc does not yet support applying capabilities, so run as root and
    # let bluetoothd drop uid to bluetooth with the right linux capabilities
    group bluetooth net_bt_admin misc
    disabled

# baudrate change 115200 to 1152000(Bluetooth)
service changebaudrate /system/bin/logwrapper /system/xbin/bccmd_115200 -t bcsp -d /dev/s3c2410_serial1 psset -r 0x1be 0x126e
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

#service hciattach /system/bin/logwrapper /system/bin/hciattach -n -s 1152000 /dev/s3c2410_serial1 bcsp 1152000
service hciattach /system/bin/logwrapper /system/bin/hciattach -n -s 115200 /dev/s3c2410_serial1 bcsp 115200
    user bluetooth
    group bluetooth net_bt_admin misc
```



```
        disabled

service hfag /system/bin/sdptool add --channel=10 HFAG
        user bluetooth
        group bluetooth net_bt_admin
        disabled
        oneshot

service hsag /system/bin/sdptool add --channel=11 HSAG
        user bluetooth
        group bluetooth net_bt_admin
        disabled
        oneshot

service opush /system/bin/sdptool add --channel=12 OPUSH
        user bluetooth
        group bluetooth net_bt_admin
        disabled
        oneshot

service pbap /system/bin/sdptool add --channel=19 PBAP
        user bluetooth
        group bluetooth net_bt_admin
        disabled
        oneshot
```

在上述代码中，每一个 service 后面列出了一种 Android 服务。

22.2.3　管理蓝牙电源

在 Android 系统的目录 system/bluetooth/中实现了 libbluedroid。

我们可以调用 rfkill 接口来控制电源管理，如果已经实现了 rfkill 接口，则无须再进行配置。如果在文件 init.rc 中已经实现了 hciattach 服务，则说明在 libbluedroid 中已经实现了对其调用工作，这样可以操作实现蓝牙的初始化工作。

22.3　低功耗蓝牙协议栈详解

从 Android 4.2 版本开始，Google 就更换了 Android 的蓝牙协议栈，从 BlueZ 换成 BlueDroid。从 Android 4.3 版本开始，提供了对蓝牙 4.0 BLE 的支持。本节将详细讲解 Android 系统中的蓝牙 4.0 BLE 的基本知识。

22.3.1　低功耗蓝牙协议栈基础

为了确保 Android 系统可以更好地支持蓝牙 4.0 BLE，Broadcom 公司特意推出了适应于 Android 平台的开源低功耗蓝牙协议栈 BlueDroid，其开发文档和 API 是开源代码，在地址 https://github.com/briandbl/framework 中保存。

在上述开源代码中，低功耗蓝牙 API 支持 Android 平台上的低功耗蓝牙通信功能。通过使用 BlueDroid 协议栈，Android 应用程序可以枚举、发现并访问低功耗蓝牙的外部设备，并且实现了低功耗蓝牙规范。

从 Android 4.2 版本开始，低功耗蓝牙模块的整体结构如图 22-3 所示。

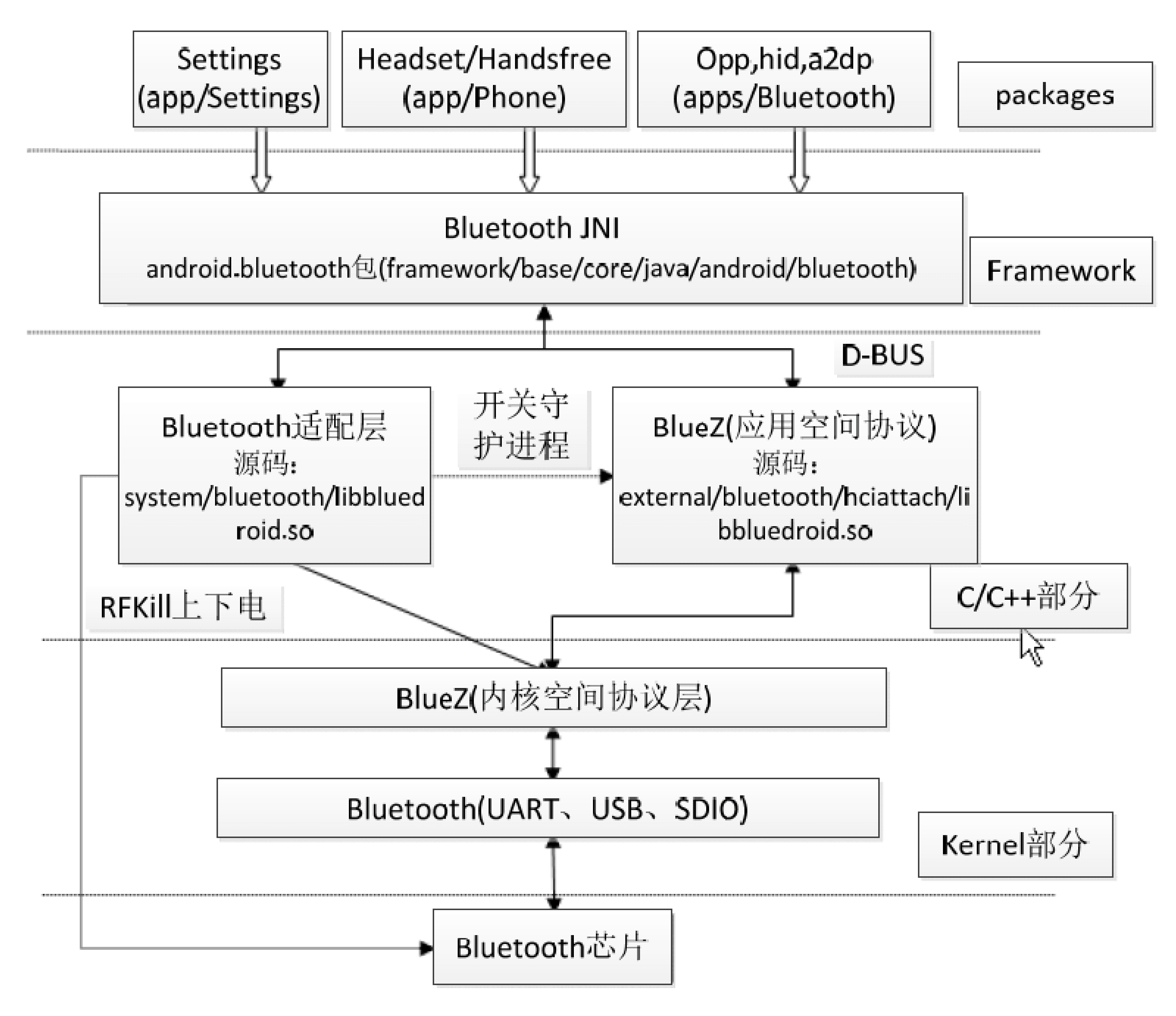


图 22-3　低功耗蓝牙模块的整体结构

注意： 虽然从 Android 4.2 版本开始，JNI 部分的代码在 packages 层中实现，但是为了便于读者从视觉上更加容易接受，所以将 JNI 部分绘制在了 Framework 层中。

22.3.2　低功耗蓝牙 API 详解

Broadcom 公司推出的低功耗蓝牙协议栈 BlueDroid 的开发文档和 API 是开源代码，被保存在地址 https://github.com/briandbl/framework 中。

下面将详细讲解主要 API 的基本功能和具体原理。

（1）本地蓝牙适配器设备

本功能不是由 Broadcom 公司提供的，而是由 Android SDK 提供的，实现源码位于目录 framework/base/core/java/android.bluetooth/BluetoothAdapter.java 下。

文件 BluetoothAdapter.java 实现了所有蓝牙交互的入口。通过使用类 BluetoothAdapter 可以实现如下功能。

☑　发现其他的蓝牙设备，查询匹配的设备集。

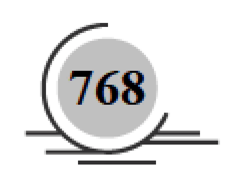
☑　使用一个已知蓝牙地址初始化蓝牙设备 BluetoothDevice。

☑　创建一个能够监听其他设备通信的类 BluetoothSocket。

（2）请求远程蓝牙设备

本功能不是由 Broadcom 公司提供的，而是由 Android SDK 提供的，源码位于目录 framework/base/core/java/android.bluetooth/BluetoothDevice.java 中。

文件 BluetoothDevice.java 代表一个远程蓝牙设备，可以支持 BLE 低功耗设备、BR/EDR 设备或 Dual-mode 类型的设备。通过使用类 BluetoothDevice 可以实现如下功能。



☑　请求获取远程蓝牙设备的连接。

☑　查询获取远程蓝牙设备的名称、地址、类和连接状态。

（3）实现客户端的低功耗蓝牙规范

在 Broadcom 公司提供的源码中，文件 BleClientProfile.java 的功能是实现客户端的低功耗蓝牙规范。在应用中要想访问远程设备中的低功耗蓝牙规范，就必须继承于类 BleClientProfile，并且需要提供要访问规范的必需参数和服务标识。通过 BleClientProfile 的派生类可以发起一个远程设备的连接，并且一个 BleClientProfile 类可能会包含多个 BleClientService 对象的实例。

（4）创建一个代表客户端角色设备上的低功耗蓝牙服务派生类

在 Broadcom 公司提供的源码中，文件 BleClientService.java 的功能是定义一个派生类，此派生类代表了客户端角色设备上的低功耗蓝牙服务。通过这个派生类可以允许应用程序读写低功耗蓝牙服务的特征，并在特征改变时注册通知。

（5）定义服务器端角色的低功耗规范

在 Broadcom 公司提供的源码中，文件 BleServerProfile.java 的功能是定义了服务器端角色的低功耗规范，在创建一个新的低功耗规范之前，需要先继承于这个类，并提供标识要访问规范所必需的参数和服务。通常来说，一个 BleServerProfile 派生的类包含一个或多个 BleServerService 对象。在 BleServerProfile 派生的类中，包含低功耗规范中定义服务的 BleServerService 对象的集合。

（6）创建低功耗服务

在 Broadcom 公司提供的源码中，文件 BleServerService.java 的功能是创建一个低功耗服务，这是服务器端角色上的低功耗规范的一部分。BleServerService 的派生类包含了一个或多个 BleCharacteristic 对象。在应用程序中，需要重写类 BleServerService 来实现一个服务。

（7）描述低功耗蓝牙服务的特性

在 Broadcom 公司提供的源码中，文件 BleCharacteristic.java 的功能是描述低功耗蓝牙服务的特性。在特性中包含了描述符、实际值和元数据，提供了表现格式或便于阅读值的描述。

（8）低功耗描述符

在 Broadcom 公司提供的源码中，文件 BleDescriptor.java 是 BleCharacteristic 的一部分，功能是定义了一个低功耗描述符。

（9）标识低功耗蓝牙规范、服务和特性

在 Broadcom 公司提供的源码中，文件 BleGattID.java 的功能是定义了一个标识低功耗蓝牙规范、服务和特性的类，此类使用 16 位或 128 位的 UUIDs 来标识一个给定的低功耗蓝牙实体，这个实体包含规范、服务和特性。

（10）为远程蓝牙设备提供额外信息

在 Broadcom 公司提供的源码中，文件 BleAdapter.java 的功能是为远程蓝牙设备提供额外的信息，能够判断远程设备是否是低功耗设备、BR/EDR 传统蓝牙设备或双模设备（同时支持低功耗和传统设备）。

（11）保存和 GATT 相关的常量

在 Broadcom 公司提供的源码中，文件 BleConstants.java 的功能是定义保存各种和 GATT 相关的常量，这些常量用于表示各种实现低功耗功能函数的属性和返回值。

22.4　Android 中的 BlueDroid

本节将详细讲解 Android 源码中低功耗协议栈 BlueDroid 的具体实现原理和应用方法。

22.4.1 Android 系统中 BlueDroid 的架构

在 Android 新系统中，采用 BlueDroid 作为默认的协议栈，BlueDroid 分为如下两个部分。

☑ Bluetooth Embedded System（BTE）：实现了 BT（蓝牙）的核心功能。

☑ Bluetooth Application Layer（BTA）：用于和 Android Framework 层进行交互。

在 Android 新系统中，BT 系统服务通过 JNI 与 BT stack 进行交互，并且通过 Binder IPC 通信与应用交互，这个系统服务同时也提供给 RD 获取不同的 BT profiles。如图 22-4 所示为 BT stack 的大体结构。

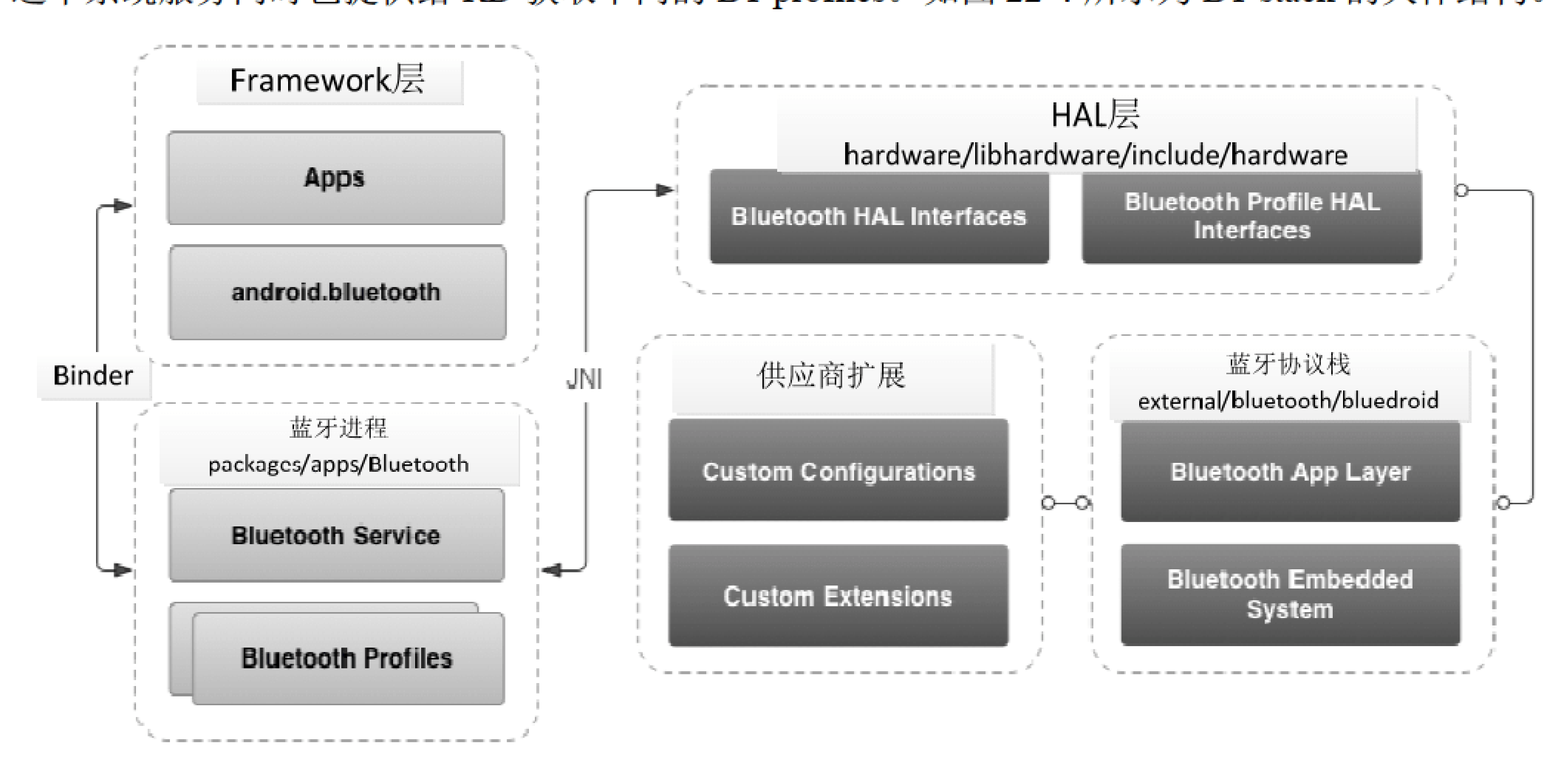


图 22-4　BT stack 的结构

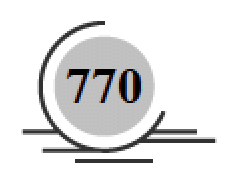
22.4.2 Application Framework 层分析

在 Application Framework 层中，通过 android.bluetooth APIS 和 Bluetooth Hardware 层进行交互，也就是通过 Binder IPC 机制调用 Bluetooth 进程。Application Framework 层的实现源码位于目录 framework/base/core/java/android.bluetooth/中。

在文件 framework/base/core/java/android/bluetooth/BluetoothA2dp.java 中定义了 connect(Bluetoothevice)方法，功能是通过 Binder IPC 通信机制调用文件 packages/apps/Bluetooth/src/com/android/bluetooth/a2dp/A2dpService.java 中的内部私有类。

文件 BluetoothA2dp.java 的具体实现代码如下。

```
public final class BluetoothA2dp implements BluetoothProfile {
    private static final String TAG = "BluetoothA2dp";
    private static final boolean DBG = true;
    private static final boolean VDBG = false;
    @SdkConstant(SdkConstantType.BROADCAST_INTENT_ACTION)
    public static final String ACTION_CONNECTION_STATE_CHANGED =
        "android.bluetooth.a2dp.profile.action.CONNECTION_STATE_CHANGED";
    @SdkConstant(SdkConstantType.BROADCAST_INTENT_ACTION)
    public static final String ACTION_PLAYING_STATE_CHANGED =
        "android.bluetooth.a2dp.profile.action.PLAYING_STATE_CHANGED";
    public static final int STATE_PLAYING    =    10;
```



```
    public static final int STATE_NOT_PLAYING    =   11;

    private Context mContext;
    private ServiceListener mServiceListener;
    private IBluetoothA2dp mService;
    private BluetoothAdapter mAdapter;

    final private IBluetoothStateChangeCallback mBluetoothStateChangeCallback =
            new IBluetoothStateChangeCallback.Stub() {
                public void onBluetoothStateChange(boolean up) {
                    if (DBG) Log.d(TAG, "onBluetoothStateChange: up=" + up);
                    if (!up) {
                        if (VDBG) Log.d(TAG,"Unbinding service...");
                        synchronized (mConnection) {
                            try {
                                mService = null;
                                mContext.unbindService(mConnection);
                            } catch (Exception re) {
                                Log.e(TAG,"",re);
                            }
                        }
                    } else {
                        synchronized (mConnection) {
                            try {
                                if (mService == null) {
                                    if (VDBG) Log.d(TAG,"Binding service...");
                                    if (!mContext.bindService(new Intent(IBluetoothA2dp.class.getName()),
mConnection, 0)) {
                                        Log.e(TAG, "Could not bind to Bluetooth A2DP Service");
                                    }
                                }
                            } catch (Exception re) {
                                Log.e(TAG,"",re);
                            }
                        }
                    }
                }
        };
BluetoothA2dp(Context context, ServiceListener l) {
        mContext = context;
        mServiceListener = l;
        mAdapter = BluetoothAdapter.getDefaultAdapter();
        IBluetoothManager mgr = mAdapter.getBluetoothManager();
        if (mgr != null) {
            try {
                mgr.registerStateChangeCallback(mBluetoothStateChangeCallback);
            } catch (RemoteException e) {
                Log.e(TAG,"",e);
            }
        }
```

```
            if (!context.bindService(new Intent(IBluetoothA2dp.class.getName()), mConnection, 0)) {
                Log.e(TAG, "Could not bind to Bluetooth A2DP Service");
            }
        }

void close() {
            mServiceListener = null;
            IBluetoothManager mgr = mAdapter.getBluetoothManager();
            if (mgr != null) {
                try {
                    mgr.unregisterStateChangeCallback(mBluetoothStateChangeCallback);
                } catch (Exception e) {
                    Log.e(TAG,"",e);
                }
            }

            synchronized (mConnection) {
                if (mService != null) {
                    try {
                        mService = null;
                        mContext.unbindService(mConnection);
                    } catch (Exception re) {
                        Log.e(TAG,"",re);
                    }
                }
            }
        }

        public void finalize() {
            close();
        }
        public boolean connect(BluetoothDevice device) {
            if (DBG) log("connect(" + device + ")");
            if (mService != null && isEnabled() &&
                isValidDevice(device)) {
                try {
                    return mService.connect(device);
                } catch (RemoteException e) {
                    Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                    return false;
                }
            }
            if (mService == null) Log.w(TAG, "Proxy not attached to service");
            return false;
        }
        public boolean disconnect(BluetoothDevice device) {
            if (DBG) log("disconnect(" + device + ")");
            if (mService != null && isEnabled() &&
                isValidDevice(device)) {
                try {
                    return mService.disconnect(device);
```

```
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return false;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return false;
    }

    public List<BluetoothDevice> getConnectedDevices() {
        if (VDBG) log("getConnectedDevices()");
        if (mService != null && isEnabled()) {
            try {
                return mService.getConnectedDevices();
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return new ArrayList<BluetoothDevice>();
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return new ArrayList<BluetoothDevice>();
    }
    public List<BluetoothDevice> getDevicesMatchingConnectionStates(int[] states) {
        if (VDBG) log("getDevicesMatchingStates()");
        if (mService != null && isEnabled()) {
            try {
                return mService.getDevicesMatchingConnectionStates(states);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return new ArrayList<BluetoothDevice>();
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return new ArrayList<BluetoothDevice>();
    }
    public int getConnectionState(BluetoothDevice device) {
        if (VDBG) log("getState(" + device + ")");
        if (mService != null && isEnabled()
            && isValidDevice(device)) {
            try {
                return mService.getConnectionState(device);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return BluetoothProfile.STATE_DISCONNECTED;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return BluetoothProfile.STATE_DISCONNECTED;
    }
    public boolean setPriority(BluetoothDevice device, int priority) {
        if (DBG) log("setPriority(" + device + ", " + priority + ")");
```

```
        if (mService != null && isEnabled()
            && isValidDevice(device)) {
            if (priority != BluetoothProfile.PRIORITY_OFF &&
                priority != BluetoothProfile.PRIORITY_ON){
              return false;
            }
            try {
                return mService.setPriority(device, priority);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return false;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return false;
    }
    public int getPriority(BluetoothDevice device) {
        if (VDBG) log("getPriority(" + device + ")");
        if (mService != null && isEnabled()
            && isValidDevice(device)) {
            try {
                return mService.getPriority(device);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return BluetoothProfile.PRIORITY_OFF;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return BluetoothProfile.PRIORITY_OFF;
    }
    public boolean isA2dpPlaying(BluetoothDevice device) {
        if (mService != null && isEnabled()
            && isValidDevice(device)) {
            try {
                return mService.isA2dpPlaying(device);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return false;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return false;
    }
    public boolean shouldSendVolumeKeys(BluetoothDevice device) {
        if (isEnabled() && isValidDevice(device)) {
            ParcelUuid[] uuids = device.getUuids();
            if (uuids == null) return false;

            for (ParcelUuid uuid: uuids) {
                if (BluetoothUuid.isAvrcpTarget(uuid)) {
                    return true;
```

```
                }
            }
        }
        return false;
    }
    public static String stateToString(int state) {
        switch (state) {
        case STATE_DISCONNECTED:
            return "disconnected";
        case STATE_CONNECTING:
            return "connecting";
        case STATE_CONNECTED:
            return "connected";
        case STATE_DISCONNECTING:
            return "disconnecting";
        case STATE_PLAYING:
            return "playing";
        case STATE_NOT_PLAYING:
          return "not playing";
        default:
            return "<unknown state " + state + ">";
        }
    }

    private ServiceConnection mConnection = new ServiceConnection() {
        public void onServiceConnected(ComponentName className, IBinder service) {
            if (DBG) Log.d(TAG, "Proxy object connected");
            mService = IBluetoothA2dp.Stub.asInterface(service);

            if (mServiceListener != null) {
                mServiceListener.onServiceConnected(BluetoothProfile.A2DP, BluetoothA2dp.this);
            }
        }
        public void onServiceDisconnected(ComponentName className) {
            if (DBG) Log.d(TAG, "Proxy object disconnected");
            mService = null;
            if (mServiceListener != null) {
                mServiceListener.onServiceDisconnected(BluetoothProfile.A2DP);
            }
        }
    };
```

在上述代码中，定义了 A2dpService 对象 service，并调用 getService()方法。A2dpService 是一个继承于类 ProfileService 的子类，而 ProfileService 是继承于类 Service 的子类。文件 A2dpService.java 的主要实现代码如下。

```
public class A2dpService extends ProfileService {
    private static final boolean DBG = false;
    private static final String TAG="A2dpService";

    private A2dpStateMachine mStateMachine;
    private Avrcp mAvrcp;
```

```
private static A2dpService sAd2dpService;

protected String getName() {
    return TAG;
}

protected IProfileServiceBinder initBinder() {
    return new BluetoothA2dpBinder(this);
}

protected boolean start() {
    mStateMachine = A2dpStateMachine.make(this, this);
    mAvrcp = Avrcp.make(this);
    setA2dpService(this);
    return true;
}

protected boolean stop() {
    mStateMachine.doQuit();
    mAvrcp.doQuit();
    return true;
}

protected boolean cleanup() {
    if (mStateMachine!= null) {
        mStateMachine.cleanup();
    }
    if (mAvrcp != null) {
        mAvrcp.cleanup();
        mAvrcp = null;
    }
    clearA2dpService();
    return true;
}

//API Methods

public static synchronized A2dpService getA2dpService(){
    if (sAd2dpService != null && sAd2dpService.isAvailable()) {
        if (DBG) Log.d(TAG, "getA2DPService(): returning " + sAd2dpService);
        return sAd2dpService;
    }
    if (DBG)   {
        if (sAd2dpService == null) {
            Log.d(TAG, "getA2dpService(): service is NULL");
        } else if (!(sAd2dpService.isAvailable())) {
            Log.d(TAG,"getA2dpService(): service is not available");
        }
    }
    return null;
```

```
    }

    private static synchronized void setA2dpService(A2dpService instance) {
        if (instance != null && instance.isAvailable()) {
            if (DBG) Log.d(TAG, "setA2dpService(): set to: " + sAd2dpService);
            sAd2dpService = instance;
        } else {
            if (DBG)   {
                if (sAd2dpService == null) {
                    Log.d(TAG, "setA2dpService(): service not available");
                } else if (!sAd2dpService.isAvailable()) {
                    Log.d(TAG,"setA2dpService(): service is cleaning up");
                }
            }
        }
    }

    private static synchronized void clearA2dpService() {
        sAd2dpService = null;
    }

    public boolean connect(BluetoothDevice device) {
        enforceCallingOrSelfPermission(BLUETOOTH_ADMIN_PERM,
                                       "Need BLUETOOTH ADMIN permission");

        if (getPriority(device) == BluetoothProfile.PRIORITY_OFF) {
            return false;
        }

        int connectionState = mStateMachine.getConnectionState(device);
        if (connectionState == BluetoothProfile.STATE_CONNECTED ||
            connectionState == BluetoothProfile.STATE_CONNECTING) {
            return false;
        }

        mStateMachine.sendMessage(A2dpStateMachine.CONNECT, device);
        return true;
    }

    boolean disconnect(BluetoothDevice device) {
        enforceCallingOrSelfPermission(BLUETOOTH_ADMIN_PERM,
                                       "Need BLUETOOTH ADMIN permission");
        int connectionState = mStateMachine.getConnectionState(device);
        if (connectionState != BluetoothProfile.STATE_CONNECTED &&
            connectionState != BluetoothProfile.STATE_CONNECTING) {
            return false;
        }

        mStateMachine.sendMessage(A2dpStateMachine.DISCONNECT, device);
        return true;
    }
```

由此可见，在接下来的通信过程中通过 Binder IPC 通信机制调用了文件 A2dpService.java 中的 connect (BluetoothDevice)方法。上述过程就是 Bluetooth Application Framework 与 Bluetooth Process 之间的调用过程。

22.4.3 分析 Bluetooth System Service 层

在 Android 系统中，Bluetooth System Service 位于 packages/apps/Bluetooth 目录下，将其打包成一个 Android App（Android 应用程序）包，并且在 Android Framework 层实现 BT Service 和各种 profile。BT App 接下来会通过 JNI 调用到 HAL 层。

在文件 A2dpService.java 中，connect 方法会发送一个 StateMachine.sendMessage(A2dpStateMachine. CONNECT, device)的 message（信息 0），这个 message 会被 A2dpStateMachine 对象的 processMessage (Message)方法接收到，对应代码如下所示。

```
case CONNECT:
                BluetoothDevice device = (BluetoothDevice) message.obj;
                broadcastConnectionState(device, BluetoothProfile.STATE_CONNECTING,
                                BluetoothProfile.STATE_DISCONNECTED);

                if (!connectA2dpNative(getByteAddress(device)) ) {
                    broadcastConnectionState(device, BluetoothProfile.STATE_DISCONNECTED,
                                BluetoothProfile.STATE_CONNECTING);
                    break;
                }

                synchronized (A2dpStateMachine.this) {
                    mTargetDevice = device;
                    transitionTo(mPending);
                }
                // TODO(BT) remove CONNECT_TIMEOUT when the stack
                // sends back events consistently
                sendMessageDelayed(CONNECT_TIMEOUT, 30000);
                break;
```

在上述代码中，会通过“connectA2dpNative(getByteAddress(device);”代码行设置通过 JNI 调用到 Native（本地程序）。

```
private native boolean connectA2dpNative(byte[] address);
```

22.4.4 JNI 层详解

在 Android 系统中，和 Bluetooth 有关的 JNI 代码位于目录 packages/apps/bluetooth/jni 中。

JNI 层的代码会调用到 HAL 层，并且在确信一些 BT 操作被触发时从 HAL 获取一些回调，如当 BT 设备被发现时。例如在 A2dp 连接的例子中，BT System Service 会通过 JNI 调用文件 com_android_bluetooth_a2dp.cpp 中的方法，此文件的主要实现代码如下。

```
namespace android {
static jmethodID method_onConnectionStateChanged;
static jmethodID method_onAudioStateChanged;

static const btav_interface_t *sBluetoothA2dpInterface = NULL;
static jobject mCallbacksObj = NULL;
static JNIEnv *sCallbackEnv = NULL;
```

```
static bool checkCallbackThread() {
    sCallbackEnv = getCallbackEnv();
    //}

    JNIEnv* env = AndroidRuntime::getJNIEnv();
    if (sCallbackEnv != env || sCallbackEnv == NULL) return false;
    return true;
}

static void bta2dp_connection_state_callback(btav_connection_state_t state, bt_bdaddr_t* bd_addr) {
    jbyteArray addr;

    ALOGI("%s", __FUNCTION__);

    if (!checkCallbackThread()) {                                              \
        ALOGE("Callback: '%s' is not called on the correct thread", __FUNCTION__); \
        return;                                                                \
    }
    addr = sCallbackEnv->NewByteArray(sizeof(bt_bdaddr_t));
    if (!addr) {
        ALOGE("Fail to new jbyteArray bd addr for connection state");
        checkAndClearExceptionFromCallback(sCallbackEnv, __FUNCTION__);
        return;
    }

    sCallbackEnv->SetByteArrayRegion(addr, 0, sizeof(bt_bdaddr_t), (jbyte*) bd_addr);
    sCallbackEnv->CallVoidMethod(mCallbacksObj, method_onConnectionStateChanged, (jint) state,
                                 addr);
    checkAndClearExceptionFromCallback(sCallbackEnv, __FUNCTION__);
    sCallbackEnv->DeleteLocalRef(addr);
}

static void bta2dp_audio_state_callback(btav_audio_state_t state, bt_bdaddr_t* bd_addr) {
    jbyteArray addr;

    ALOGI("%s", __FUNCTION__);

    if (!checkCallbackThread()) {                                              \
        ALOGE("Callback: '%s' is not called on the correct thread", __FUNCTION__); \
        return;                                                                \
    }
    addr = sCallbackEnv->NewByteArray(sizeof(bt_bdaddr_t));
    if (!addr) {
        ALOGE("Fail to new jbyteArray bd addr for connection state");
        checkAndClearExceptionFromCallback(sCallbackEnv, __FUNCTION__);
        return;
    }

    sCallbackEnv->SetByteArrayRegion(addr, 0, sizeof(bt_bdaddr_t), (jbyte*) bd_addr);
    sCallbackEnv->CallVoidMethod(mCallbacksObj, method_onAudioStateChanged, (jint) state,
```

```
                                            addr);
    checkAndClearExceptionFromCallback(sCallbackEnv, __FUNCTION__);
    sCallbackEnv->DeleteLocalRef(addr);
}

static btav_callbacks_t sBluetoothA2dpCallbacks = {
    sizeof(sBluetoothA2dpCallbacks),
    bta2dp_connection_state_callback,
    bta2dp_audio_state_callback
};

static void classInitNative(JNIEnv* env, jclass clazz) {
    int err;
    const bt_interface_t* btInf;
    bt_status_t status;

    method_onConnectionStateChanged =
        env->GetMethodID(clazz, "onConnectionStateChanged", "(I[B)V");

    method_onAudioStateChanged =
        env->GetMethodID(clazz, "onAudioStateChanged", "(I[B)V");
    /*
    if ( (btInf = getBluetoothInterface()) == NULL) {
        ALOGE("Bluetooth module is not loaded");
        return;
    }

    if ( (sBluetoothA2dpInterface = (btav_interface_t *)
          btInf->get_profile_interface(BT_PROFILE_ADVANCED_AUDIO_ID)) == NULL) {
        ALOGE("Failed to get Bluetooth A2DP Interface");
        return;
    }
    ALOGI("%s: succeeds", __FUNCTION__);
}

static void initNative(JNIEnv *env, jobject object) {
    const bt_interface_t* btInf;
    bt_status_t status;

    if ( (btInf = getBluetoothInterface()) == NULL) {
        ALOGE("Bluetooth module is not loaded");
        return;
    }

    if (sBluetoothA2dpInterface !=NULL) {
         ALOGW("Cleaning up A2DP Interface before initializing...");
         sBluetoothA2dpInterface->cleanup();
         sBluetoothA2dpInterface = NULL;
    }

    if (mCallbacksObj != NULL) {
```

```
            ALOGW("Cleaning up A2DP callback object");
            env->DeleteGlobalRef(mCallbacksObj);
            mCallbacksObj = NULL;
    }

    if ( (sBluetoothA2dpInterface = (btav_interface_t *)
            btInf->get_profile_interface(BT_PROFILE_ADVANCED_AUDIO_ID)) == NULL) {
        ALOGE("Failed to get Bluetooth A2DP Interface");
        return;
    }

    if ( (status = sBluetoothA2dpInterface->init(&sBluetoothA2dpCallbacks)) != BT_STATUS_SUCCESS) {
        ALOGE("Failed to initialize Bluetooth A2DP, status: %d", status);
        sBluetoothA2dpInterface = NULL;
        return;
    }

    mCallbacksObj = env->NewGlobalRef(object);
}

static void cleanupNative(JNIEnv *env, jobject object) {
    const bt_interface_t* btInf;
    bt_status_t status;

    if ( (btInf = getBluetoothInterface()) == NULL) {
        ALOGE("Bluetooth module is not loaded");
        return;
    }

    if (sBluetoothA2dpInterface !=NULL) {
        sBluetoothA2dpInterface->cleanup();
        sBluetoothA2dpInterface = NULL;
    }

    if (mCallbacksObj != NULL) {
        env->DeleteGlobalRef(mCallbacksObj);
        mCallbacksObj = NULL;
    }
}

static jboolean connectA2dpNative(JNIEnv *env, jobject object, jbyteArray address) {
    jbyte *addr;
    bt_bdaddr_t * btAddr;
    bt_status_t status;

    ALOGI("%s: sBluetoothA2dpInterface: %p", __FUNCTION__, sBluetoothA2dpInterface);
    if (!sBluetoothA2dpInterface) return JNI_FALSE;

    addr = env->GetByteArrayElements(address, NULL);
    btAddr = (bt_bdaddr_t *) addr;
    if (!addr) {
```

```
        jniThrowIOException(env, EINVAL);
        return JNI_FALSE;
    }

    if ((status = sBluetoothA2dpInterface->connect((bt_bdaddr_t *)addr)) != BT_STATUS_SUCCESS) {
        ALOGE("Failed HF connection, status: %d", status);
    }
    env->ReleaseByteArrayElements(address, addr, 0);
    return (status == BT_STATUS_SUCCESS) ? JNI_TRUE : JNI_FALSE;
}

static jboolean disconnectA2dpNative(JNIEnv *env, jobject object, jbyteArray address) {
    jbyte *addr;
    bt_status_t status;

    if (!sBluetoothA2dpInterface) return JNI_FALSE;

    addr = env->GetByteArrayElements(address, NULL);
    if (!addr) {
        jniThrowIOException(env, EINVAL);
        return JNI_FALSE;
    }

    if ( (status = sBluetoothA2dpInterface->disconnect((bt_bdaddr_t *)addr)) != BT_STATUS_SUCCESS) {
        ALOGE("Failed HF disconnection, status: %d", status);
    }
    env->ReleaseByteArrayElements(address, addr, 0);
    return (status == BT_STATUS_SUCCESS) ? JNI_TRUE : JNI_FALSE;
}

static JNINativeMethod sMethods[] = {
    {"classInitNative", "()V", (void *) classInitNative},
    {"initNative", "()V", (void *) initNative},
    {"cleanupNative", "()V", (void *) cleanupNative},
    {"connectA2dpNative", "([B)Z", (void *) connectA2dpNative},
    {"disconnectA2dpNative", "([B)Z", (void *) disconnectA2dpNative},
};

int register_com_android_bluetooth_a2dp(JNIEnv* env)
{
    return jniRegisterNativeMethods(env, "com/android/bluetooth/a2dp/A2dpStateMachine",
                                    sMethods, NELEM(sMethods));
}

}
```

在上述代码中用到了结构体对象 sBluetoothA2dpInterface，此对象在方法 initNative(JNIEnv*env, jobject object)中定义获取，代码如下所示。

```
if ( (sBluetoothA2dpInterface = (btav_interface_t *)
      btInf->get_profile_interface(BT_PROFILE_ADVANCED_AUDIO_ID)) == NULL) {
    ALOGE("Failed to get Bluetooth A2DP Interface");
```

```
    return;
}
```

22.4.5　HAL 硬件抽象层详解

硬件抽象层用于定义 android.bluetooth APIs 和 BT process 调用的标准接口，BT HAL 的头文件如下。

- ☑ hardware/libhardware/include/hardware/bluetooth.h
- ☑ hardware/libhardware/include/hardware/bt_*.h

JNI 中 sBluetoothA2dpInterface 是一个 btav_interface_t 结构体，在文件 hardware/libhardware/include/hardware/bt_av.h 中定义，具体实现代码如下。

```
typedef struct {
    size_t          size;
    bt_status_t (*init)( btav_callbacks_t* callbacks );
    bt_status_t (*connect)( bt_bdaddr_t *bd_addr );
    bt_status_t (*disconnect)( bt_bdaddr_t *bd_addr );
    void  (*cleanup)( void );
} btav_interface_t;
```

Android 系统新版本默认蓝牙协议栈 BlueDroid 在如下所示的目录下实现。

```
external/bluetooth/bluedroid
```

上述 stack 实现了通用的 BT HAL 并且可以通过扩展和改变配置进行自定义。例如 A2dp 的连接会调用到 external/bluetooth/bluedroid/btif/src/btif_av.c 的 connect()方法，此方法的具体实现代码如下。

```
static bt_status_t connect(bt_bdaddr_t *bd_addr)
{
    BTIF_TRACE_EVENT1("%s", __FUNCTION__);
    CHECK_BTAV_INIT();

    return btif_queue_connect(UUID_SERVCLASS_AUDIO_SOURCE, bd_addr, connect_int);
}
```

22.5　Android 蓝牙模块的运作流程

通过本章前面内容的学习，读者已经了解了 Android 系统中蓝牙模块的内核代码的基本知识。本节将以 Android 源码为蓝本，介绍 Android 4.3 系统中蓝牙模块的具体运作流程。

22.5.1　打开蓝牙设备

在 Android 系统的内置蓝牙模块中，打开蓝牙功能的开关是 settings 选项中的 switch 开关。打开后需要使用 systemServer.java 的代码开启蓝牙服务，相关代码如下所示。

```
if (SystemProperties.get("ro.kernel.qemu").equals("1")) {
                Slog.i(TAG, "No Bluetooh Service (emulator)");
            } else if (factoryTest == SystemServer.FACTORY_TEST_LOW_LEVEL) {
                Slog.i(TAG, "No Bluetooth Service (factory test)");
            } else {
                Slog.i(TAG, "Bluetooth Manager Service");
                bluetooth = new BluetoothManagerService(context);
```

```
                ServiceManager.addService(BluetoothAdapter.BLUETOOTH_MANAGER_SERVICE, bluetooth);
            }
```

在类 bluetoothManagerService 的构造方法中，方法 loadStoredNameAndAddress()用于读取蓝牙并打开默认的名称，方法 isBluetoothPersistedStateOn()用于判断是否已经打开蓝牙，如果已经打开，则后面的操作需要执行开启蓝牙的动作，前面的注册广播代码就起到了这个作用，对应代码如下所示。

```
BluetoothManagerService(Context context) {
        ...一些变量声明初始化...
        IntentFilter filter = new IntentFilter(Intent.ACTION_BOOT_COMPLETED);
        filter.addAction(BluetoothAdapter.ACTION_LOCAL_NAME_CHANGED);
        filter.addAction(Intent.ACTION_USER_SWITCHED);
        registerForAirplaneMode(filter);
        mContext.registerReceiver(mReceiver, filter);
        loadStoredNameAndAddress();
        if (isBluetoothPersistedStateOn()) {
            mEnableExternal = true;
        }
    }
```

返回到开关界面，界面开关实现文件是 BluetoothEnabler.java，而方法 setBluetoothEnabled()是界面开关的具体实现，对应代码如下所示。

```
public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        // Show toast message if Bluetooth is not allowed in airplane mode
        if (isChecked &&
                !WirelessSettings.isRadioAllowed(mContext, Settings.Global.RADIO_BLUETOOTH)) {
            Toast.makeText(mContext, R.string.wifi_in_airplane_mode, Toast.LENGTH_SHORT).show();
            // Reset switch to off
            buttonView.setChecked(false);
        }

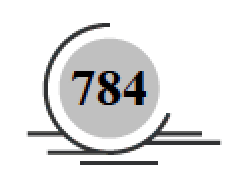
        if (mLocalAdapter != null) {
            mLocalAdapter.setBluetoothEnabled(isChecked);
        }
        mSwitch.setEnabled(false);
    }
```

在上述代码中判断是否是飞行模式，如果是飞行模式，则会弹出 toast 提示，由阅读方法 setBluetoothEnabled()可知，mLocalAdapter(LocalBluetoothAdapter)只是个过渡方法，里面的 mAdapter(BluetoothAdapter)方法才是真正的主角，对应代码如下。

```
public void setBluetoothEnabled(boolean enabled) {
    boolean success = enabled    ? mAdapter.enable() : mAdapter.disable();

    if (success) {
        setBluetoothStateInt(enabled
            ? BluetoothAdapter.STATE_TURNING_ON
            : BluetoothAdapter.STATE_TURNING_OFF);
    } else {
        ...
    }
}
```

文件 BluetoothAdapter.java 实现了一个单例模式的应用，提供了供其他程序调用蓝牙的一些方法，外部程序在调用蓝牙方法之前需要用到文件 BluetoothAdapter.java，此文件中的 BluetoothAdapter 对象演示了



Android 系统典型的 binder 应用，对应代码如下。

```
public static synchronized BluetoothAdapter getDefaultAdapter() {
    if (sAdapter == null) {
        IBinder b = ServiceManager.getService(BLUETOOTH_MANAGER_SERVICE);
        if (b != null) {
            IBluetoothManager managerService = IBluetoothManager.Stub.asInterface(b);
            sAdapter = new BluetoothAdapter(managerService);
        } else {
            Log.e(TAG, "Bluetooth binder is null");
        }
    }
    return sAdapter;
}
```

调用 mAdapter.enable()方法之后，外部其他应用也需要调用 enable()方法。这里的文件 BluetoothAdapter 位于 Framework 层，文件 BluetoothAdapter.java 中的 enable()方法调用会先回到文件 BluetoothManagerService.java 中的 enable()方法，然后来到文件 BluetoothManagerService.java 中的 handleEnable()方法，后面需要跳转到新类。

```
private void handleEnable(boolean persist, boolean quietMode) {
     synchronized(mConnection) {
        if ((mBluetooth == null) && (!mBinding)) {
            //Start bind timeout and bind
            Message timeoutMsg=mHandler.obtainMessage(MESSAGE_TIMEOUT_BIND);
            mHandler.sendMessageDelayed(timeoutMsg,TIMEOUT_BIND_MS);
            mConnection.setGetNameAddressOnly(false);
            Intent i = new Intent(IBluetooth.class.getName());
            if (!mContext.bindService(i, mConnection,Context.BIND_AUTO_CREATE,
                                      UserHandle.USER_CURRENT)) {
                mHandler.removeMessages(MESSAGE_TIMEOUT_BIND);
                Log.e(TAG, "Fail to bind to: " + IBluetooth.class.getName());
            } else {
                mBinding = true;
            }
        }
```

分析如下所示的 log 信息。

```
ActivityManager: Start proc com.android.bluetooth for service com.android.bluetooth/.btservice.AdapterService:"
```

在 AdapterService 服务中一共有 3 个 enable()，跳转关系非常简单，其中最后一个比较关键。

```
public synchronized boolean enable(boolean quietMode) {
    enforceCallingOrSelfPermission(BLUETOOTH_ADMIN_PERM,
        "Need BLUETOOTH ADMIN permission");
    if (DBG)debugLog("Enable called with quiet mode status =   " + mQuietmode);
    mQuietmode   = quietMode;
    Message m =
        mAdapterStateMachine.obtainMessage(AdapterState.USER_TURN_ON);
    mAdapterStateMachine.sendMessage(m);
    return true;
}
```

这样将从一个状态接受命令跳到另一个状态，因为是开启蓝牙，所以先去文件 AdapterState.java 的内部类 offstate.java 中找，在这个分支 USER_TURN_ON 中会看到方法 mAdapterService.processStart()，在这里面可以看到蓝牙遍历下所支持的 profile，最后会发出一个带 AdapterState.STARTED 标识的消息。具体处理功能在 AdapterState.java 中实现，对应代码如下。

```
case STARTED:    {
  if (DBG) Log.d(TAG,"CURRENT_STATE=PENDING, MESSAGE = STARTED, isTurningOn=" + isTurningOn
+ ", isTurningOff=" + isTurningOff);
  //Remove start timeout
  removeMessages(START_TIMEOUT);
  //Enable
  boolean ret = mAdapterService.enableNative();
  if (!ret) {
      Log.e(TAG, "Error while turning Bluetooth On");
      notifyAdapterStateChange(BluetoothAdapter.STATE_OFF);
      transitionTo(mOffState);
  } else {
      sendMessageDelayed(ENABLE_TIMEOUT, ENABLE_TIMEOUT_DELAY);
  }
```

在上述代码中使用 JNI 调用了 enableNative()函数，根据 Android JNI 的函数命名习惯可以很容易找到函数 enableNative() 对应的 C++ 函数在文件 packages/apps/Bluetooth/jni/com_android_bluetooth_btservice_AdapterService. cpp 中实现。

函数 enableNative()的具体实现代码如下。

```
static jboolean enableNative(JNIEnv* env, jobject obj) {
    ALOGV("%s:",__FUNCTION__);
    jboolean result = JNI_FALSE;
     if (!sBluetoothInterface) return result;
     int ret = sBluetoothInterface->enable();
     result = (ret == BT_STATUS_SUCCESS) ? JNI_TRUE : JNI_FALSE;
     return result;
}
```

在上述代码中，sBluetoothInterface 在文件/external/bluetooth/bluedroid/btif/src/bluetooth.c 中定义。

定义 bt_interface_t bluetoothInterface 的代码如下。

```
    static const bt_interface_t bluetoothInterface = {
    sizeof(bt_interface_t),
    init,
    enable,
    disable,
    ...
    start_discovery,
    cancel_discovery,
    create_bond,
    remove_bond,
    cancel_bond,
   ...
};
```

函数 enable()在 bluetooth.c 中实现，对应代码如下。

```
static int enable( void )
{
    ALOGI("enable");
    /* sanity check */
    if (interface_ready() == FALSE)
        return BT_STATUS_NOT_READY;
    return btif_enable_bluetooth();
}
```

在上述代码中用到了函数 bt_status_t btif_enable_bluetooth()，具体实现代码如下。

```
bt_status_t btif_enable_bluetooth(void)
{
    BTIF_TRACE_DEBUG0("BTIF ENABLE BLUETOOTH");
    if (btif_core_state != BTIF_CORE_STATE_DISABLED)
    {
        ALOGD("not disabled\n");
        return BT_STATUS_DONE;
    }
    btif_core_state = BTIF_CORE_STATE_ENABLING;
    /* Create the GKI tasks and run them */
    bte_main_enable(btif_local_bd_addr.address);
    return BT_STATUS_SUCCESS;
}
```

在上述代码中调用了函数 bte_main_enable()，此函数在文件 external/bluetooth/bluedroid/main/bte_main.c 中定义，具体实现代码如下。

```
void bte_main_enable(uint8_t *local_addr)
{
    APPL_TRACE_DEBUG1("%s", __FUNCTION__);
    ...
#if (defined (BT_CLEAN_TURN_ON_DISABLED) && BT_CLEAN_TURN_ON_DISABLED == TRUE)
        APPL_TRACE_DEBUG1("%s  Not Turninig Off the BT before Turninig ON", __FUNCTION__);
#else
        /* toggle chip power to ensure we will reset chip in case
           a previous stack shutdown wasn't completed gracefully */
        bt_hc_if->set_power(BT_HC_CHIP_PWR_OFF);
#endif
        bt_hc_if->set_power(BT_HC_CHIP_PWR_ON);
        bt_hc_if->preload(NULL);
    }
     ...
}
```

在上述代码中调用了文件 external/bluetooth/bluedroid/hci/src/bt_hci_bdroid.c 中的函数 set_power()，具体实现代码如下。

```
static void set_power(bt_hc_chip_power_state_t state)
{
    int pwr_state;

    BTHCDBG("set_power %d", state);

    /* Calling vendor-specific part */
    pwr_state = (state == BT_HC_CHIP_PWR_ON) ? BT_VND_PWR_ON : BT_VND_PWR_OFF;

    if (bt_vnd_if)
        bt_vnd_if->op(BT_VND_OP_POWER_CTRL, &pwr_state);
    else
        ALOGE("vendor lib is missing!");
}
```

在上述代码中，bt_vnd_if 源于文件 external/bluetooth/bluedroid/hci/include/bt_vendor_lib.h，对应代码如下。

```
extern const bt_vendor_interface_t BLUETOOTH_VENDOR_LIB_INTERFACE;
bt_vendor_interface_t *bt_vnd_if=NULL;
```

Google 已经定义好了接口，具体实现需要由 Vendor 厂商来完成，这部分代码需要看各家芯片商而言，这部分代码通常不会公开，打开蓝牙功能需要用如下类似的字符串实现。

```
static const char* BT_DRIVER_MODULE_PATH = "/system/lib/modules/mbt8xxx.ko";
static const char* BT_DRIVER_MODULE_NAME = "bt8xxx";
static const char* BT_DRIVER_MODULE_INIT_ARG = " init_cfg=";
static const char* BT_DRIVER_MODULE_INIT_CFG_PATH = "bt_init_cfg.conf";
```

还有类似下面的动作，insmod 加载驱动，rfkill 控制上下电，不同厂商的具体做法会有所不同。

```
ret = insmod(BT_DRIVER_MODULE_PATH, arg_buf);
ret = system("/system/bin/rfkill block all");
```

到此为止，打开 Android 蓝牙的流程介绍全部结束。对于 Vendor 部分的代码需要看各自厂商的代码，通常在开启蓝牙后才会通电，这样比较符合逻辑并节省电量。查看是否通电的方法是：连上手机，用 adb shell 看 sys/class/rfkill 目录下的 state 的状态值，有些厂商会把蓝牙和 WiFi 的上电算在一起，这一点需要特别注意，以避免误判。

22.5.2 搜索蓝牙

在 Android 系统中，有如下两种启动蓝牙搜索工作的方法。

☑ 在蓝牙设置界面开启蓝牙，这样会直接开始搜索。

☑ 先打开蓝牙开关，在进入蓝牙设置界面也会触发搜索。

上述两种方式在最后都要来到文件 BluetoothSettings.java 中的方法 startScanning()，此方法的实现代码如下。

```
private void updateContent(int bluetoothState, boolean scanState) {
    if (numberOfPairedDevices == 0) {
           preferenceScreen.removePreference(mPairedDevicesCategory);
           if (scanState == true) {
                 mActivityStarted = false;
                 startScanning();
          } else<span style="font-family: Arial, Helvetica, sans-serif;">       ...</span>
}
private void startScanning() {
        if (!mAvailableDevicesCategoryIsPresent) {
            getPreferenceScreen().addPreference(mAvailableDevicesCategory);
        }
        mLocalAdapter.startScanning(true);
    }
```

由此可见，蓝牙的搜索和打开流程在结构上是一致的，需要利用文件 LocalBluetoothAdapter.java 过渡到文件 BluetoothAdapter.java，然后再跳转至文件 AdapterService.java。在上述过渡过程中，startScaning()方法变成了 startDiscovery()方法，startScaning()方法在文件 packages/apps/Settings/src/com/android/settings/bluetooth/LocalBluetoothAdapter.java 中实现。

startScaning()方法的实现代码如下。

```
void startScanning(boolean force) {
if (!mAdapter.isDiscovering()) {
        if (!force) {
            // Don't scan more than frequently than SCAN_EXPIRATION_MS,
            // unless forced
            if (mLastScan + SCAN_EXPIRATION_MS > System.currentTimeMillis()) {
                return;
            }
```

```
            // If we are playing music, don't scan unless forced.
            A2dpProfile a2dp = mProfileManager.getA2dpProfile();
            if (a2dp != null && a2dp.isA2dpPlaying()) {
                return;
            }
        }
//这里才是我们最关注的，前面限制条件关注一下就行了
        if (mAdapter.startDiscovery()) {
            mLastScan = System.currentTimeMillis();
        }
}
```

方法 startDiscovery()在文件 frameworks/base/core/java/android/bluetooth/BluetoothAdapter.java 中定义，具体代码如下。

```
public boolean startDiscovery() {
    ...
    AdapterService service = getService();
    if (service == null) return false;
    return service.startDiscovery();
}
```

在上述代码中，service.startDiscovery()在文件 packages/apps/Bluetooth/src/com/android/bluetooth/btservice/AdapterService.java 中定义。

方法 service.startDiscovery()的具体实现代码如下。

```
boolean startDiscovery() {
    enforceCallingOrSelfPermission(BLUETOOTH_ADMIN_PERM,
            "Need BLUETOOTH ADMIN permission");

    return startDiscoveryNative();
}
```

接下来分析 JNI 文件 packages/apps/Bluetooth/jni/com_android_bluetooth_btservice_AdapterService.cpp，对应代码如下。

```
static jboolean startDiscoveryNative(JNIEnv* env, jobject obj) {
    ALOGV("%s:",__FUNCTION__);

    jboolean result = JNI_FALSE;
    if (!sBluetoothInterface) return result;

    int ret = sBluetoothInterface->start_discovery();
    result = (ret == BT_STATUS_SUCCESS) ? JNI_TRUE : JNI_FALSE;
    return result;
}
```

在上述代码中，函数 start_discovery()在文件 external/bluetooth/bluedroid/btif/src/bluetooth.c 中定义，具体实现代码如下。

```
static int start_discovery(void)
{
    /* sanity check */
    if (interface_ready() == FALSE)
        return BT_STATUS_NOT_READY;

    return btif_dm_start_discovery();
}
```

在上述代码中，函数 btif_dm_start_discovery()在文件 external/bluetooth/bluedroid/btif/src/btif_dm.c 中定义，具体实现代码如下。

```
bt_status_t   btif_dm_start_discovery(void)
{
    tBTA_DM_INQ inq_params;
    tBTA_SERVICE_MASK services = 0;

    BTIF_TRACE_EVENT1("%s", __FUNCTION__);
    /* TODO: Do we need to handle multiple inquiries at the same time? */

    /* Set inquiry params and call API */
#if (BLE_INCLUDED == TRUE)
    inq_params.mode = BTA_DM_GENERAL_INQUIRY|BTA_BLE_GENERAL_INQUIRY;
#else
    inq_params.mode = BTA_DM_GENERAL_INQUIRY;
#endif
    inq_params.duration = BTIF_DM_DEFAULT_INQ_MAX_DURATION;

    inq_params.max_resps = BTIF_DM_DEFAULT_INQ_MAX_RESULTS;
    inq_params.report_dup = TRUE;

    inq_params.filter_type = BTA_DM_INQ_CLR;
    /* TODO: Filter device by BDA needs to be implemented here */

    /* Will be enabled to TRUE once inquiry busy level has been received */
    btif_dm_inquiry_in_progress = FALSE;
    /* find nearby devices */
    BTA_DmSearch(&inq_params, services, bte_search_devices_evt);

    return BT_STATUS_SUCCESS;
}
```

在上述代码中，bte_search_devices_evt()函数是核心，此函数的主要实现代码如下。

```
  static void bte_search_devices_evt(tBTA_DM_SEARCH_EVT event, tBTA_DM_SEARCH *p_data)
{
          UINT16 param_len = 0;

          if (p_data)
              param_len += sizeof(tBTA_DM_SEARCH);
          /* Allocate buffer to hold the pointers (deep copy). The pointers will point to the end of the tBTA_
DM_SEARCH */
          switch (event)
          {
              case BTA_DM_INQ_RES_EVT:
              {
                  if (p_data->inq_res.p_eir)
                      param_len += HCI_EXT_INQ_RESPONSE_LEN;
              }
              break;
             ...
          }
```

```
            BTIF_TRACE_DEBUG3("%s event=%s param_len=%d", __FUNCTION__, dump_dm_search_event
(event), param_len);

            /* if remote name is available in EIR, set teh flag so that stack doesnt trigger RNR */
            if (event == BTA_DM_INQ_RES_EVT)
                p_data->inq_res.remt_name_not_required = check_eir_remote_name(p_data, NULL, NULL);

            btif_transfer_context (btif_dm_search_devices_evt , (UINT16) event, (void *)p_data, param_len,
                (param_len > sizeof(tBTA_DM_SEARCH)) ? search_devices_copy_cb : NULL);
      }
```

函数 btif_dm_search_devices_evt()用于在界面展示搜索蓝牙的结果，此函数在文件 external/bluetooth/bluedroid/btif/src/btif_dm.c 中定义，具体实现代码如下。

```
static void btif_dm_search_devices_evt (UINT16 event, char *p_param)
{
    tBTA_DM_SEARCH *p_search_data;
    BTIF_TRACE_EVENT2("%s event=%s", __FUNCTION__, dump_dm_search_event(event));

    switch (event)
    {
        case BTA_DM_DISC_RES_EVT:
        {
            p_search_data = (tBTA_DM_SEARCH *)p_param;
            /* Remote name update */
            if (strlen((const char *) p_search_data->disc_res.bd_name))
            {
                bt_property_t properties[1];
                bt_bdaddr_t bdaddr;
                bt_status_t status;

                properties[0].type = BT_PROPERTY_BDNAME;
                properties[0].val = p_search_data->disc_res.bd_name;
                properties[0].len = strlen((char *)p_search_data->disc_res.bd_name);
                bdcpy(bdaddr.address, p_search_data->disc_res.bd_addr);

                status = btif_storage_set_remote_device_property(&bdaddr, &properties[0]);
                ASSERTC(status == BT_STATUS_SUCCESS, "failed to save remote device property",
status);
                HAL_CBACK(bt_hal_cbacks, remote_device_properties_cb,
                                 status, &bdaddr, 1, properties);
            }
            /* TODO: Services? */
        }
        break;
```

函数 BTA_DmSearch()的功能是发送消息，在文件 external/bluetooth/bluedroid/bta/dm/bta_dm_api.c 中定义，具体实现代码如下。

```
void BTA_DmSearch(tBTA_DM_INQ *p_dm_inq, tBTA_SERVICE_MASK services, tBTA_DM_SEARCH_
CBACK *p_cback)
{   tBTA_DM_API_SEARCH*p_msg;
    if ((p_msg = (tBTA_DM_API_SEARCH *) GKI_getbuf(sizeof(tBTA_DM_API_SEARCH))) != NULL)
    {
```

```
            memset(p_msg, 0, sizeof(tBTA_DM_API_SEARCH));

            p_msg->hdr.event = BTA_DM_API_SEARCH_EVT;
            memcpy(&p_msg->inq_params, p_dm_inq, sizeof(tBTA_DM_INQ));
            p_msg->services = services;
            p_msg->p_cback = p_cback;
            p_msg->rs_res    = BTA_DM_RS_NONE;
            bta_sys_sendmsg(p_msg);
      }
}
```

方法 deviceFoundCallback()最后会来到文件/packages/apps/Bluetooth/src/com/android/bluetooth/btservice/RemoteDevices.java 中，具体实现代码如下。

```
void deviceFoundCallback(byte[] address) {
            // The device properties are already registered - we can send the intent
            // now
            BluetoothDevice device = getDevice(address);
            debugLog("deviceFoundCallback: Remote Address is:" + device);
            DeviceProperties deviceProp = getDeviceProperties(device);
            if (deviceProp == null) {
                  errorLog("Device Properties is null for Device:" + device);
                  return;
            }

            Intent intent = new Intent(BluetoothDevice.ACTION_FOUND);
            intent.putExtra(BluetoothDevice.EXTRA_DEVICE, device);
            intent.putExtra(BluetoothDevice.EXTRA_CLASS,
                        new BluetoothClass(Integer.valueOf(deviceProp.mBluetoothClass)));
            intent.putExtra(BluetoothDevice.EXTRA_RSSI, deviceProp.mRssi);
            intent.putExtra(BluetoothDevice.EXTRA_NAME, deviceProp.mName);

            mAdapterService.sendBroadcast(intent, mAdapterService.BLUETOOTH_PERM);
      }
```

这样通过界面发送广播，在应用层中会通过 handle 将收到的广播显示出来，这个 handle 可以在文件 BluetoothEventManager.java 的构造函数中找到，具体代码如下。

```
addHandler(BluetoothDevice.ACTION_FOUND, new DeviceFoundHandler());
      private final BroadcastReceiver mBroadcastReceiver = new BroadcastReceiver() {
      @Override
      public void onReceive(Context context, Intent intent) {
            String action = intent.getAction();
            BluetoothDevice device = intent
                  .getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);

            Handler handler = mHandlerMap.get(action);
            if (handler != null) {
                  handler.onReceive(context, intent, device);
            }
      }
};
```

上述 handle 和 DeviceFoundHandler 相对应，具体代码如下。

```
private class DeviceFoundHandler implements Handler {
      public void onReceive(Context context, Intent intent,
```

```
                BluetoothDevice device) {
        ...
        // TODO Pick up UUID. They should be available for 2.1 devices
        // Skip for now, there's a bluez problem and we are not getting uuids even for 2.1
        CachedBluetoothDevice cachedDevice = mDeviceManager.findDevice(device);
        if (cachedDevice == null) {
            cachedDevice = mDeviceManager.addDevice(mLocalAdapter, mProfileManager, device);
            Log.d(TAG, "DeviceFoundHandler created new CachedBluetoothDevice: "
                    + cachedDevice);
            // callback to UI to create Preference for new device
            dispatchDeviceAdded(cachedDevice);
        }
       ...
    }
}
```

在上述 if 语句中，方法 dispatchDeviceAdded()用于向界面分发消息并处理消息，并使用文件/packages/apps/Settings/src/com/android/settings/bluetooth/DeviceListPreferenceFragment.java 实现界面显示功能，对应代码如下。

```
public void onDeviceAdded(CachedBluetoothDevice cachedDevice) {
    if (mDevicePreferenceMap.get(cachedDevice) != null) {
        return;
    }

    // Prevent updates while the list shows one of the state messages
    if (mLocalAdapter.getBluetoothState() != BluetoothAdapter.STATE_ON) return;

    if (mFilter.matches(cachedDevice.getDevice())) {
        createDevicePreference(cachedDevice);
    }
 }
```

上述代码的最后一个分支是界面显示需要做的工作，整个过程从 Settings 界面开始再到 Settings 界面显示搜索到蓝牙结束，对应代码如下。

```
void createDevicePreference(CachedBluetoothDevice cachedDevice) {
    BluetoothDevicePreference preference = new BluetoothDevicePreference(
            getActivity(), cachedDevice);

    initDevicePreference(preference);
    mDeviceListGroup.addPreference(preference);
    mDevicePreferenceMap.put(cachedDevice, preference);
}
```

到目前为止包括前面的打开流程分析，仅是针对代码流程做的分析，对于蓝牙协议方面还没有涉及。

22.5.3 传输 OPP 文件

在 Android 系统中，OPP 文件是蓝牙传输的文件。在分享蓝牙文件的过程中，使用的是蓝牙应用 OPP 目录下的代码。在 Android 设备中当使用蓝牙发送文件时，发送端先来到文件 packages/apps/Bluetooth/src/com/android/bluetooth/opp/BluetoothOppLauncherActivity.java 处，这是一个没有界面只是提取文件信息的中转站文件，此文件的核心是两个分支 action.equals(Intent.ACTION_SEND)和 action.equals(Intent.ACTION_SEND_MULTIPLE)，对应代码如下。

```
if (action.equals(Intent.ACTION_SEND) || action.equals(Intent.ACTION_SEND_MULTIPLE)) {
            //Check if Bluetooth is available in the beginning instead of at the end
            if (!isBluetoothAllowed()) {
                Intent in = new Intent(this, BluetoothOppBtErrorActivity.class);
                in.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                in.putExtra("title", this.getString(R.string.airplane_error_title));
                in.putExtra("content", this.getString(R.string.airplane_error_msg));
                startActivity(in);
                finish();
                return;
            }
            if (action.equals(Intent.ACTION_SEND)) {
            ...
               Thread t = new Thread(new Runnable() {
                            public void run() {
                                BluetoothOppManager.getInstance(BluetoothOppLauncherActivity.this)
                                    .saveSendingFileInfo(type,fileUri.toString(), false);
                                //Done getting file info..Launch device picker
                                //and finish this activity
                                launchDevicePicker();
                                finish();
                            }
                        });   ...
            } else if (action.equals(Intent.ACTION_SEND_MULTIPLE)) {
              ...
            }
```

在上述代码中，前面的 isBluetoothAllowed()方法会判断是否处于飞行模式，如果是，则禁止发送 OPP 文件。在 launchDevicePicker()中会通过条件语句(!BluetoothOppManager.getInstance(this).isEnabled())判断蓝牙是否已经打开，如果已经打开则进入设备选择界面 DeviceListPreferenceFragment(DevicePickerFragment)选择设备。在这个跳转过程中，"new Intent(BluetoothDevicePicker.ACTION_LAUNCH)"字符串的完整定义如下。

```
public static final String ACTION_LAUNCH = "android.bluetooth.devicepicker.action.LAUNCH";
```

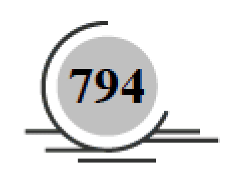
对应的文件是 frameworks/base/core/java/android/bluetooth/BluetoothDevicePicker.java，在 setting 应用的 manifest.xml 文件中会发现，对应代码如下。

```
<activity android:name=".bluetooth.DevicePickerActivity"
                android:theme="@android:style/Theme.Holo.DialogWhenLarge"
                android:label="@string/device_picker"
                android:clearTaskOnLaunch="true">
            <intent-filter>
                <action android:name="android.bluetooth.devicepicker.action.LAUNCH" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
```

上述代码指向了 DevicePickerActivity，其源码路径是 packages/apps/Settings/src/com/android/settings/bluetooth/DevicePickerActivity.java。

类 DevicePickerActivity 的实现代码很简单，只有一个 onCreate，并且只在里面加载了一个布局文件 bluetooth_device_picker.xml，对应代码如下。

```
<fragment
        android:id="@+id/bluetooth_device_picker_fragment"
        android:name="com.android.settings.bluetooth.DevicePickerFragment"
```



```
            android:layout_width="match_parent"
            android:layout_height="0dip"
            android:layout_weight="1" />
```

上述布局文件设置了下一步的位置是 DevicePickerFragment，此时可以看到配对后的蓝牙列表，列表中的 sendDevicePickedIntent 又会发送一个广播，对应代码如下。

```
void onDevicePreferenceClick(BluetoothDevicePreference btPreference) {
    mLocalAdapter.stopScanning();
    LocalBluetoothPreferences.persistSelectedDeviceInPicker(
            getActivity(), mSelectedDevice.getAddress());
    if ((btPreference.getCachedDevice().getBondState() ==
            BluetoothDevice.BOND_BONDED) || !mNeedAuth) {
        sendDevicePickedIntent(mSelectedDevice);
        finish();
    } else {
        super.onDevicePreferenceClick(btPreference);
    }
}<div>      public static final String ACTION_LAUNCH = "android.bluetooth.devicepicker.action.LAUNCH";
     private void sendDevicePickedIntent(BluetoothDevice device) {
     Intent intent = new Intent(BluetoothDevicePicker.ACTION_DEVICE_SELECTED);
     intent.putExtra(BluetoothDevice.EXTRA_DEVICE, device);
     if (mLaunchPackage != null && mLaunchClass != null) {
         intent.setClassName(mLaunchPackage, mLaunchClass);
     }
    getActivity().sendBroadcast(intent); }
```

通过 BluetoothDevicePicker.ACTION_DEVICE_SELECTED 查找，会在文件 packages/apps/Bluetooth/src/com/android/bluetooth/opp/BluetoothOppReceiver.java 中找到对上述发送广播的处理，对应代码如下。

```
else if (action.equals(BluetoothDevicePicker.ACTION_DEVICE_SELECTED)) {
            BluetoothOppManager mOppManager = BluetoothOppManager.getInstance(context);
            BluetoothDevice remoteDevice = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);

            // Insert transfer session record to database
            mOppManager.startTransfer(remoteDevice);

            // Display toast message
            String deviceName = mOppManager.getDeviceName(remoteDevice);
            ...
}
```

在上述代码中，调用的方法 mOppManager.startTransfer(remoteDevice)在文件 packages/apps/Bluetooth/src/com/android/bluetooth/opp/BluetoothOppManager.java 中实现，功能是开启线程执行发送动作，通过 run()方法分单个或多个文件进行发送处理，其中单个文件的发送代码如下。

```
public void startTransfer(BluetoothDevice device) {
    if (V) Log.v(TAG, "Active InsertShareThread number is : " + mInsertShareThreadNum);
    InsertShareInfoThread insertThread;
    synchronized (BluetoothOppManager.this) {
        if (mInsertShareThreadNum > ALLOWED_INSERT_SHARE_THREAD_NUMBER) {
            ...
            return;
            }
            insertThread = new InsertShareInfoThread(device, mMultipleFlag, mMimeTypeOfSendingFile,
```

```
                    mUriOfSendingFile, mMimeTypeOfSendingFiles, mUrisOfSendingFiles,
                    mIsHandoverInitiated);
                if (mMultipleFlag) {
                    mfileNumInBatch = mUrisOfSendingFiles.size();
                }
            }
            insertThread.start();
        }
        public void run() {
                Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
                ...
                if (mIsMultiple) {
                    insertMultipleShare();
                } else {
                    insertSingleShare();
                }
                ... }
```

以 insertSingleShare()方法的实现过程为例，调用了方法 mContext.getContentResolver().insert，此方法在文件 bluetooth/src/com/android/Bluetooth/opp/BluetoothOppProvider.java 中定义，具体代码如下。

```
public Uri insert(Uri uri, ContentValues values) {
if (rowID != -1) {
    context.startService(new Intent(context, BluetoothOppService.class));
    ret = Uri.parse(BluetoothShare.CONTENT_URI + "/" + rowID);
    context.getContentResolver().notifyChange(uri, null);
} else {
    if (D) Log.d(TAG, "couldn't insert into btopp database");
}
```

由上述代码可知，又调用了 BluetoothOppService 服务，实现文件是 packages/apps/Bluetooth/ src/com/android/bluetooth/opp/BluetoothOppService.java。在文件 BluetoothOppService.java 的方法 onStartCommand 中会看到 updateFromProvider()方法，通过此方法开启了一个 UpdateThread 线程，并通过 run()方法执行到 BluetoothOppTransfer.java 的对象，对应代码如下。

```
private void insertShare(Cursor cursor, int arrayPos) {
    ...
    if (info.isReadyToStart()) {
        ...
        if (mBatchs.size() == 0) {
            ...
            mBatchs.add(newBatch);
            if (info.mDirection == BluetoothShare.DIRECTION_OUTBOUND) {
                mTransfer = new BluetoothOppTransfer(this, mPowerManager, newBatch);
            } else if (info.mDirection == BluetoothShare.DIRECTION_INBOUND) {
                mServerTransfer = new BluetoothOppTransfer(this, mPowerManager, newBatch,
                        mServerSession);
            }
            if (info.mDirection == BluetoothShare.DIRECTION_OUTBOUND && mTransfer != null) {
                mTransfer.start();
            } else if (info.mDirection == BluetoothShare.DIRECTION_INBOUND
                    && mServerTransfer != null) {
                mServerTransfer.start();
            }
```

```
            } else {
                ...
        }}
```

方法 start()并不是线程方法，其实现文件是 packages/apps/Bluetooth/src/com/android/bluetooth/opp/BluetoothOppTransfer.java。

对应的代码如下。

```
public void start() {
        ...这里省略未贴的代码是检查蓝牙是否打开，提高了安全性
        if (mHandlerThread == null) {
            ...
            if (mBatch.mDirection == BluetoothShare.DIRECTION_OUTBOUND) {
                /* for outbound transfer, we do connect first */
                startConnectSession();
            } else if (mBatch.mDirection == BluetoothShare.DIRECTION_INBOUND) {
                startObexSession();
            }
        }
}
```

上述代码用于发送文件和接收文件，如果分享给别人则是 OUTBOUND，会先执行 startConnectSession()；如果接收文件则直接执行 startObexSession()，startConnectSession()函数最后还是要执行到 startObexSession()处，对应代码如下。

```
public static final int DIRECTION_OUTBOUND = 0;
 // This transfer is inbound, e.g. receive file from other device.
 public static final int DIRECTION_INBOUND = 1;
```

发送文件功能在 BluetoothOppObexClientSession.java 中开启，对应代码如下。

```
private void startObexSession() {
        if (mBatch.mDirection == BluetoothShare.DIRECTION_OUTBOUND) {
            if (V) Log.v(TAG, "Create Client session with transport " + mTransport.toString());
            mSession = new BluetoothOppObexClientSession(mContext, mTransport);
        } else if (mBatch.mDirection == BluetoothShare.DIRECTION_INBOUND) {
            if (mSession == null) {
                markBatchFailed();
                mBatch.mStatus = Constants.BATCH_STATUS_FAILED;
                return;
            }
            if (V) Log.v(TAG, "Transfer has Server session" + mSession.toString());
        }
        mSession.start(mSessionHandler);
        processCurrentShare();
  }
```

真正的发送文件功能在文件 packages/apps/Bluetooth/src/com/android/bluetooth/opp/BluetoothOppObexClientSession.java 中实现。

对应代码如下所示。

```
private void doSend() {
        int status = BluetoothShare.STATUS_SUCCESS;
        ...省略关于 status 值的判断
        if (status == BluetoothShare.STATUS_SUCCESS) {
            /* do real send */ //看到这个注释了吧，它才是真正的 sendFile
            if (mFileInfo.mFileName != null) {
```

```
                status = sendFile(mFileInfo);
            } else {
                /* this is invalid request */
                status = mFileInfo.mStatus;
            }
            waitingForShare = true;
        } else {
            Constants.updateShareStatus(mContext1, mInfo.mId, status);
        }
        if (status == BluetoothShare.STATUS_SUCCESS) {
            Message msg = Message.obtain(mCallback);
            msg.what = BluetoothOppObexSession.MSG_SHARE_COMPLETE;
            msg.obj = mInfo;
            msg.sendToTarget();
        } else {
            Message msg = Message.obtain(mCallback);
            msg.what = BluetoothOppObexSession.MSG_SESSION_ERROR;
            mInfo.mStatus = status;
            msg.obj = mInfo;
            msg.sendToTarget();
        }
    }
}
```

当执行完 sendFile 后会把分享成功或失败的消息传回去，在 sendFile 中会执行打包的过程，对于各个字段的具体说明需要分析文件 frameworks/base/obex/javax/obex/HeaderSet.java，到此为止，蓝牙发送文件的基本流程讲解完毕。在蓝牙接收文件的过程中会收到 MSG_INCOMING_BTOPP_CONNECTION 消息，这是因为在蓝牙打开时（即蓝牙状态是 BluetoothAdapter.STATE_ON 时）会执行 startSocketListener()方法，在此函数中开启了监听程序，对应代码如下所示。

```
private void createServerSession(ObexTransport transport) {
    mServerSession = new BluetoothOppObexServerSession(this, transport);
    mServerSession.preStart();
}
```

对于蓝牙接收文件部分的流程和发送流程原理完全一样，在此不再详细分析。读者要想更好地理解蓝牙 OPP 文件的传输过程，需要了解 OBEX 基础协议方面的知识。